Naval Research Laboratory
Washington, DC 20375-5320

NRL/MR/5540--04-8841

A Taxonomy of Software Deceptive Interpretation in the Linux Operating System

Jim Luo
Margery Li
Amitabh Khashnobish
John McDermott
Judy Froscher

Center for High Assurance Computer Systems
Information Technology Division

November 10, 2004

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
10-11-2004	Memorandum	

4. TITLE AND SUBTITLE

A Taxonomy of Software Deceptive Interpretation in the Linux Operating System

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER
63235N

6. AUTHOR(S)

Jim Luo, Margery Li, Amitabh Khashnobish, John McDermott, and Judy Froscher

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER
55-8089-W4

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Naval Research Laboratory, Code 5540
4555 Overlook Avenue, SW
Washington, DC 20375-5320

8. PERFORMING ORGANIZATION REPORT NUMBER

NRL/MR/5540--04-8841

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Office of Naval Research
Ralph Wachter, Code 311
800 North Quincy Street
Arlington, VA 22217-5660

10. SPONSOR / MONITOR'S ACRONYM(S)

11. SPONSOR / MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION / AVAILABILITY STATEMENT

Approved for public release; distribution is unlimited.

13. SUPPLEMENTARY NOTES

14. ABSTRACT

Rootkits are malicious tools installed on compromised computer systems that help intruders take advantage of and maintain unauthorized access. Modern rootkits routinely employ *deceptive interpretation* to evade detection. This allows them to remain hidden and operational for extended periods of time, drastically prolonging and escalating the damage from the system compromise. This report investigates the concept of deceptive interpretation in order to explore high assurance approaches to detect rootkits. A taxonomy was developed through a systematic analysis of the Linux operating system that enumerates *all* possible mechanisms of performing software deceptive interpretation. Many novel mechanisms, not yet implemented in published rootkits, were discovered and included in the taxonomy. Categorizaton was based on the system objects that need to be modified for the deceptive interpretation mechanism. As a result, detectors that target the set of system objects associated with a category will be able to detect all deceptive interpreters in that category including previously unknown implementations. This work can serve as the basis for developing an alternative to the signature-based approach with the capability to provide *categorical protection* against deceptive interpreters and rootkits.

15. SUBJECT TERMS

Rootkits; Deceptive interpretation; Taxonomy; Detection; Loadable kernal module

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Jim Luo
Unclassified	Unclassified	Unclassified	UL	56	19b. TELEPHONE NUMBER *(include area code)* (202) 404-8764

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std. Z39.18

Table of Content

Table of Figures

A TOXONOMY OF SOFTWARE DECEPTIVE INTERPRETATION IN THE LINUX OPERATING SYSTEM

Abstract

Rootkits are malicious tools installed on compromised computer systems that help intruders take advantage of and maintain unauthorized access. Modern rootkits routinely employ *deceptive interpretation* to evade detection. This allows them to remain hidden and operational for extended periods of time, drastically prolonging and escalating the damage from the system compromise. State-of-the-art tools for detecting these sophisticated rootkits are haphazard and inadequate. Their signature-based approaches are unable to cope with novel or individualized rootkits and they will not scale well as rootkits proliferate both in numbers and sophistication. This paper investigates the concept of deceptive interpretation in order to explore high assurance approaches to detect rootkits. A taxonomy was developed through a systematic analysis of the Linux operating system that enumerates *all* possible mechanisms of performing software deceptive interpretation. Many novel mechanisms, not yet implemented in published rootkits, were discovered and included in the taxonomy. Categorization was based on the system objects that need to be modified for the deceptive interpretation mechanism. As a result, detectors that target the set of system objects associated with a category will be able to detect all deceptive interpreters in that category including previously unknown implementations. This work can serve as the basis for developing an alternative to the signature-based approach with the capability to provide *categorical protection* against deceptive interpreters and rootkits.

Manuscript approved October 4, 2004.

1. Introduction

Computer systems are constantly under the threat of intrusion. This threat is frequently realized especially in commodity products. No matter how tightly a computer is locked down, there is always at least one flaw that attackers can use to break in. In 2003 alone, 137,529 incidents were reported to CERT/CC and the number is growing exponentially each year dating back to 1990 [1]. However, intrusion is only the beginning of a compromise. After the attackers gain root access to the system, they can install rootkits containing malicious software including backdoors, sniffers, and tools to hide their presence on the system. These tools will run with root privilege and have the ability to control the system completely. However, backdoors and sniffers tend to have rather large signatures that could be easily detected. What makes rootkits exceptionally dangerous is their ability to hide from detection through deceptive interpretation. Published and widely available rootkits routinely contain code that actively hides their. Deceptive interpretation can fool both automated tools and human system administrators into thinking their systems are safe. They enable a rootkit and its malicious payload to operate for an extended period of time thus drastically prolonging the system compromise and escalating the damage.

Currently, nothing is available for Linux systems that could provide a high level of protection against deceptive interpretation. Rootkits and the tools for detecting them are in an arms race. Detection tools evolve only in response to developments in the specific rootkits they need to detect. In such an arms race, the attacker has the initiative and rootkit designers are able to remain a step ahead of the detectors. As a result, new rootkits employing methods never before seen will always evade detection. In fact, most new rootkits are developed specifically to defeat detection tools available at the time. The best system administrators can possibly do with the means publicly available to them is to say their systems do not contain specific rootkits within the scope of their detection tools. They have no assurance against new rootkits or custom rootkits created by individual hackers. This means a single successful intrusion could result in a system becoming compromised indefinitely. With the suitable rootkit installed, there is nothing the system administrator can do on a running system to detect the compromise or the initial intrusion. This situation is totally unacceptable, especially for critical military computer systems.

This paper presents a complete taxonomy of deceptive interpretation in the Linux operating system. This work contributes to developing an alternative approach to detect deceptive interpretation and rootkits. Instead of scanning for the presence of specific rootkits through signatures, the absence of all rootkits can be confirmed by verifying that malicious modifications to the system for deceptive interpretation have not been made. This integrity-verification approach, based on a comprehensive taxonomy, can provide categorical evidence regarding all possible deceptive interpreters and lead to a much higher level of assurance that systems are not tainted by deceptive interpretation and rootkits.

The following sections are organized as follows. Section 2 provides an overview of rootkits in general. Section 3 focuses on the deceptive interpreter aspect of rootkits and defines the concept of deceptive interpretation. Section 4 outlines specifics of the Linux operating system design that make it susceptible to rootkits. Section 5 presents the taxonomy of deceptive interpreters and discusses each category in detail. Section 6 presents a survey of several existing rootkits and their place within the taxonomy. Sections 7 and 8 describe some of the countermeasures currently available against rootkits, along with their limitations. Section 9 discusses the implications of the taxonomy and how it could be used to create detectors for deceptive interpretation and rootkits that are superior to what is currently available.

2. Rootkits

A clear and consistent definition of a rootkit is lacking in current computer security literature. For the purpose of this paper, a *rootkit* is defined as a toolkit installed on a root-compromised machine that exhibits at least one of the following behaviors:

1. Performs automated malicious activities. This is a wide category that includes malicious software such as network sniffers, keystroke loggers, worms and viruses.
2. Allows unauthorized access to the system in the future. This would include the installation of backdoors, modification of login databases and disabling of security services that perform access control.
3. Prevents the detection of the intrusion. This is the part of a rootkit that aims to sanitize the system and erase evidence of the initial break-in.
4. Prevents detection of the rootkit. This component hides the presence of the rootkit itself on the system and enables it to function for an extended period of time.

The first two categories of behaviors can be thought of as the payloads of the rootkit. They are the components that actually take advantage of the system compromise to perform malicious tasks. The last two categories are intended to keep the payloads operational for extended periods of time by evading detection. The fourth category, preventing the detection of the rootkit itself, is usually implemented through a deceptive interpreter.

It is important to note that rootkits are not exploits. Rootkits are not used for obtaining root access to a host, rather they are used to maintain and take advantage of such access. They are installed only after the system is already compromised. There are a number of ways to gain unauthorized root access on a system. Prepackaged exploits for root privilege are widely available on the web for Linux systems [2, 3]. Keeping the operating system and applications up to date in terms of security patches will defeat many exploits. However, this is rarely the case in the real world and systems remain vulnerable. Even if systems are fully patched, they will still be vulnerable to 0-day attacks that exploit brand new vulnerabilities [4]. Root passwords can also be stolen through social engineering, or through attacks such as network sniffers or keystroke loggers. Moreover, the attack could originate from an insider with legitimate access to the system and a root account. It must be recognized that system compromises are impossible to prevent completely and there will always be opportunities to install rootkits.

The first program that could be classified as a rootkit appeared more than a decade ago in 1994 [5]. The CERT advisory described a malicious program installed on root compromised machines that collected host and user authentication information by sniffing traffic on the local area network. These first rootkits only contained payloads. They made no attempts to provide for future access, or to hide the system compromise. Subsequent rootkits included Trojan horse login programs that provided backdoors for attackers to revisit previously compromised systems. Soon after, rootkits evolved to include features that made them truly dangerous – the ability to hide.

The first Linux rootkits appeared back in 1994, the same year the operating system itself was released [6]. The first complete Linux rootkit, *Linux Rootkit II version 1.0*, was released in 1996 [6]. It exhibited all four of the defining behaviors of rootkits. It had a network packet sniffer, a login Trojan that granted root privileges to anyone presenting a magic password, a log sanitizer to erase traces of the initial intrusion, and a number of binary Trojan horses that actively concealed the presence of the rootkit on a system.

Since then, advances in rootkit design have moved them into the kernel. Instead of simply altering the executable binaries, modern rootkits are able to subvert the underlying operating system on which all user space programs rely. For example, a rootkit could alter the system calls related to file access and hide specific files from all user space programs. As a consequence, rootkits became much harder to detect. Since the binaries are not modified, checking the integrity of these files is useless. Furthermore, placing rootkits in kernel space makes them much more powerful. They are now able to interact with the system at a low level with full privilege and bypass any restrictions put in place by the operating system.

The open source nature of the Linux operating system makes it especially susceptible to kernel rootkit attacks. Assuming the source code for proprietary closed source operating systems such as Windows is unavailable, rootkit writers have to guess at the inner workings of the system. This is security through obscurity and not a robust or desirable protection scheme. Closed source operating systems are likely to have many more vulnerabilities than Linux. In the practical sense, however, this restriction on information means fewer people get to know about vulnerabilities directly from the source-code, which does make writing rootkits for such platforms much harder. Rootkits in general do not rely on software bugs, but rather take advantage of the particular way the system is designed to function. Armed with the source code, authors of Linux rootkits can systematically analyze the operating system to find ways of exploiting it.

The expertise needed to write rootkits is well within reach of the average programmer familiar with Linux and operating systems programming. A Black Hat seminar titled "Aspects of Offensive Rootkit Technology" claims to be able to train anyone with rudimentary programming skills to create their own rootkits that perform deceptive interpretation from scratch in only two days [7]. Information and tutorials on writing rootkits are widely available on the internet to anyone [8]. The number of people capable of writing rootkits is significant. Rootkits are relatively small programs that rarely exceed several hundred lines. The actual meat of a rootkit, the key concept that allows it to perform deceptive interpretation, is usually coded in a few dozen lines. Understanding these few lines of code will allow other programmers to create new rootkits using the same mechanism. They can incorporate new capabilities to suit a particular need, or make general refinements for a better rootkit. Rootkit development is a highly modular and additive process. Components of rootkits rarely need to interact with each other. They are more like separate tools in the same toolkit rather than a tightly coupled program. Breakthroughs in the field can be quickly incorporated into rootkits with little or no modification to existing components.

For those who are incapable or unwilling to write their own rootkits, prepackaged ones are also widely available on the Internet [2, 9]. Many of these packages are sophisticated, feature rich, automated, and user friendly with an easy-to-understand interface. Technical skill is not required to actually deploy these rootkits. For the most part, they can be compiled out of the box and work on the target system without any modification.

The danger of rootkits lies in their ability to stay hidden and operate for extended periods of time. Modern rootkits employ sophisticated techniques to conceal their presence on the host. They actively try to cover up their tracks by concealing files, network connections and running processes, wiping logs, interfering with security processes running on the host, hiding from virus scanners and intrusion detection systems, and preventing the human system administrator from noticing their presence. The key to their ability to evade detection is deceptive interpretation. That is the aspect of rootkits this paper will focus on.

3. Deceptive Interpretation

A *deceptive interpreter* is a malicious agent on a system that is capable of observing and changing the results of computations according to a predefined policy. Deceptive interpreters are able to change one or more of the following in order to enforce their policies:

1. The inputs for the computation
2. The sequence of operations executed
3. The information returned by the computation

A necessary policy for all deceptive interpreters is that they themselves must remain hidden. If the deceptive interpreter itself is detected, the compromise of the system is automatically revealed. A deceptive interpreter is defeated as soon as it is detected. In order to hide the system compromise, the originator of the computation must be misled without realizing it.

In order to evade detection, rootkits must obscure the true state of the host from any program or human operator attempting to ascertain its integrity. The only way to do so is through some form of deceptive interpretation. It could be as simple as a Trojan binary that produces the wrong result, or as complex as a full fledged system reference monitor that mediates all operations. The targets of deceptive interpretation include anything that seeks to reveal the presence of the rootkit.

The *coverage* of a deceptive interpreter refers to the set of operations considered safe for the deceptive interpreter. No sequence of operations can be constructed entirely from the covered set of operations such that upon execution, the policy of the deceptive interpreter will be violated. The coverage of a deceptive interpreter is defined by satisfaction of three requirements on the set of covered operations.

1. Always invoked. The deceptive interpreter must mediate all operations relevant to its policy. No operation or sequence of operations in the covered set should be able to bypass deceptive interpretation.
2. Tamperproof. The deceptive interpreter must never act in a way that contradicts the defined policy in response to operations in the covered set. No operation or sequence of operations should cause the deceptive interpreter to malfunction or fail.
3. Undetectable. The deceptive interpreter itself must be undetectable by operations in the covered set. This requirement includes both direct detection of the deceptive interpreter and indirect detection such as when the deceptive interpreter returns unexpected results or when it crashes the system.

The only operations that could potentially defeat the deceptive interpreter are ones that are outside its coverage. This means first, operations that are irrelevant to the policy of the deceptive interpreter are in the covered set. For example, if the policy only pertains to hiding files, operations that are irrelevant to files will be part of the covered set by default. Second, the sophistication of the deceptive interpreter is determined by its coverage. For example, a Trojan binary file for the *ls* command can successfully mislead

operations that rely on that binary. Operations that do not rely on the *ls* command such as ones that use the *cat* command will fall outside of the coverage of that deceptive interpreter. On the other hand, if the deceptive interpreter is in the kernel and is able to mediate all file operations, then the coverage of the interpreter would include all file operations in user space. However, certain low level operations will be able to bypass the kernel and they would be considered outside the coverage of that deceptive interpreter.

These requirements are similar to those of a reference monitor in the context of access control [10, 11]. The difference is that reference monitors are security mechanisms that enforce legitimate policies on the system. Deceptive interpreters on the other hand, enforce malicious policies to the detriment of the system. Another difference is that the deceptive interpreter must not be detected. Deceptive interpreters cannot reveal their presence in the system and must provide responses that appear legitimate.

A deceptive interpreter is said to have *complete coverage* if its set of covered operations encompasses all possible operations on the system. In practice, deceptive interpreters only cover a handful of operations. The set of all possible operations on a system would include reboots from secure read-only media, or even a clean reinstallation of the operating system. It is unfathomable that a rootkit could survive such operations. Deceptive interpreters are usually not complete boxes, but rather intermittent walls that seek to enforce their own policies on a small specific set of operations. This level of coverage is usually sufficient for in the context of rootkits.

The goal of the deceptive interpreter as a part of a rootkit is to hide the presence of the rootkit on the host system. This implies hiding objects and behaviors that could signal its presence, specifically files, network connections, running processes, and the deceptive interpreter itself. Covered operations vary widely depending on the sophistication of the rootkit. They could range from covering only a few commonly used system commands such as *ls*, *ps*, and *netstat*, to potentially covering all software operations.

Currently, publicly available rootkits are very effective in terms of the first two requirements for covered operations – always invoked and tamperproof. Modern techniques place deceptive interpreters inside the kernel enabling them to take almost complete control of the host system. However, most of them are weak in terms of the third requirement and remain easily detectable. This is partly due to the culture of the rootkit authors. Most rootkit descriptions that are published include a chapter that explains how the rootkit could be detected on affected systems. Most mechanisms for deceptive interpretation can be detected through relatively intuitive and straightforward methods. For example, if a rootkit creates Trojan binaries, the obvious way of detecting this is to check the integrity of binary files.

On the other hand, highly stealthy rootkits are simply not necessary in the present computer security environment. The relatively unsophisticated methods currently employed in hiding the rootkit are sufficient for deployment on most systems. While there are obvious ways of detecting deceptive interpretation, those checks are rarely performed. System administrators are for the most part unaware of the problem of

rootkits and deceptive interpretation. They concentrate their efforts on hardening their systems against intrusions and take it for granted that compromises have not already occurred [12]. Many system administrators only start to suspect the presence of rootkits when their machines start acting erratically and crashing. However, this only happens in situations when the rootkit is broken and malfunctioning. Properly functioning rootkits do not significantly affect the normal operation of the host and system administrators would not have any hints that their machines are compromised.

As the threat of rootkits becomes better known, more and more system administrators will include checks for rootkits as part of their routine system maintenance. While this is a good thing in general, it will invariably cause rootkit writers to increase the sophistication of their deceptive interpreters to avoid detection. The technical know-how to create much better deceptive interpreters already exists. The most popular rootkits available currently are generations behind the most advanced mechanisms of deceptive interpretation proposed in papers. Unpublished rootkits that are extremely stealthy could already be in use by the Black Hat community.

The fundamental consequence of deceptive interpretation is that the host can no longer be trusted to inspect itself. If the detection system falls inside the coverage of the deceptive interpreter, it will not be able to detect the presence of the rootkit. The potential coverage of deceptive interpreters in general is impossible to predict. It is not unreasonable to assume the coverage could encompass the software operations of the detector.

All software checks could be defeated by a sufficiently advanced deceptive interpreter. Software integrity checkers operate on systems that could be tainted by the very deceptive interpreters they are trying to detect. It is impossible with current technology for software code to self-validate or communicate directly with the processor to ascertain the true state of the system. No sequence of operations could be executed that can categorically detect the presence of a deceptive interpreter. At the lowest level, if the deceptive interpreter has full control and complete knowledge of the system, it could simply not execute code that could violate its policy and return false results generated to appear legitimate. This basic strategy comes from Turing's proof of the halting problem [13]. The deceptive interpreter can identify the points in the software that contain decisions or results that violate its policy and act to neutralize them. This is indeed very hard to do. It would require intimate knowledge of the system and the security mechanisms protecting it. However, highly critical systems, especially those in the military, do not have the luxury of only dealing with probable scenarios. They need to take the hardest attacker approach in evaluating system security and take into account all possible attacks and develop safeguards against them.

The concept of deceptive interpretation must be examined from both the practical and theoretical point of view. The current signature-based approaches for detecting rootkits are haphazard and will not apply to more sophisticated forms of deceptive interpretation. Systematic understanding of deceptive interpretation is essential in creating effective countermeasures to the threat of rootkits.

The taxonomy presented in this paper will deal exclusively with software deceptive interpreters. While it is possible to perform deceptive interpretation directly on hardware, the opportunity to do so is very limited. For the most part, hardware is immutable. Commands are executed by hardwired circuits that cannot be changed. Attackers would need to physically replace the hardware with a malicious duplicate in order to perform deceptive interpretation. Some hardware devices could indeed be updated through software means to alter their behavior. For example, microcode on both Intel and AMD processors can be patched through software updates [14, 15]. Theoretically, CPU instructions could be altered directly to perform deceptive interpretation. Deceptive interpreters located in the CPU would be very powerful and extremely hard to detect. In reality, however, the software patch is only capable of making small changes to the CPU microcode. It is certainly not capable of creating a meaningful deceptive interpreter. While it is important to recognize the possibility of hardware based deceptive interpreters, they are relatively scarce in the context of rootkits.

4. Overview of the Linux Operating System

Linux provides a distinction between operations in *Kernel space* and *User space*. Kernel space is reserved for the operating system. It has the highest level of privilege and can directly access hardware devices and memory. Operations in user mode are restricted. They are prohibited from accessing the kernel and the underlying system hardware. System calls provide the interface for user space processes to interact with the kernel and utilize kernel services. Execution control is transferred from user space to kernel space through interrupts. They could either be hardware interrupts generated by hardware devices, or software interrupts generated by running processes such as when making a system call.

In the case of a hardware interrupt, the path of execution proceeds as follows in the Intel x86 architecture. The user process is halted and control is passed into the kernel space. The processor locates the interrupt descriptor table (IDT) which points to the interrupt handler routine. After the interrupt handler routine completes, execution is returned to user space, either to the same process that was running before the interrupt, or to a completely different process depending on the specific interrupt.

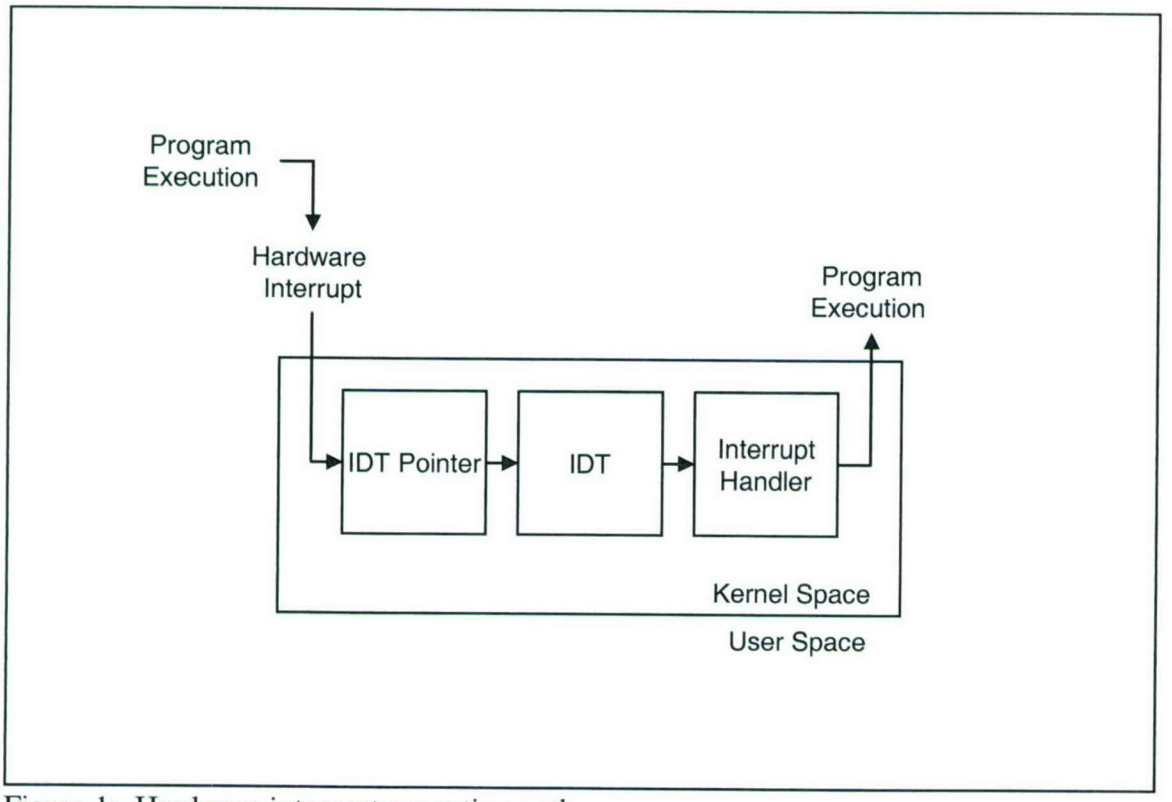

Figure 1: Hardware interrupt execution path

System calls are triggered by software interrupts generated by the currently executing user space process. From there, the execution path proceeds in much the same way as hardware interrupts. The interrupt handler routine in this case calls the system call

handler routine. The system call is parsed and the appropriate system call routine is called by consulting the system call table. The system call routine could in turn call helper routines which may be part of another system interface. After completion of the system call function, execution is returned to the user space process.

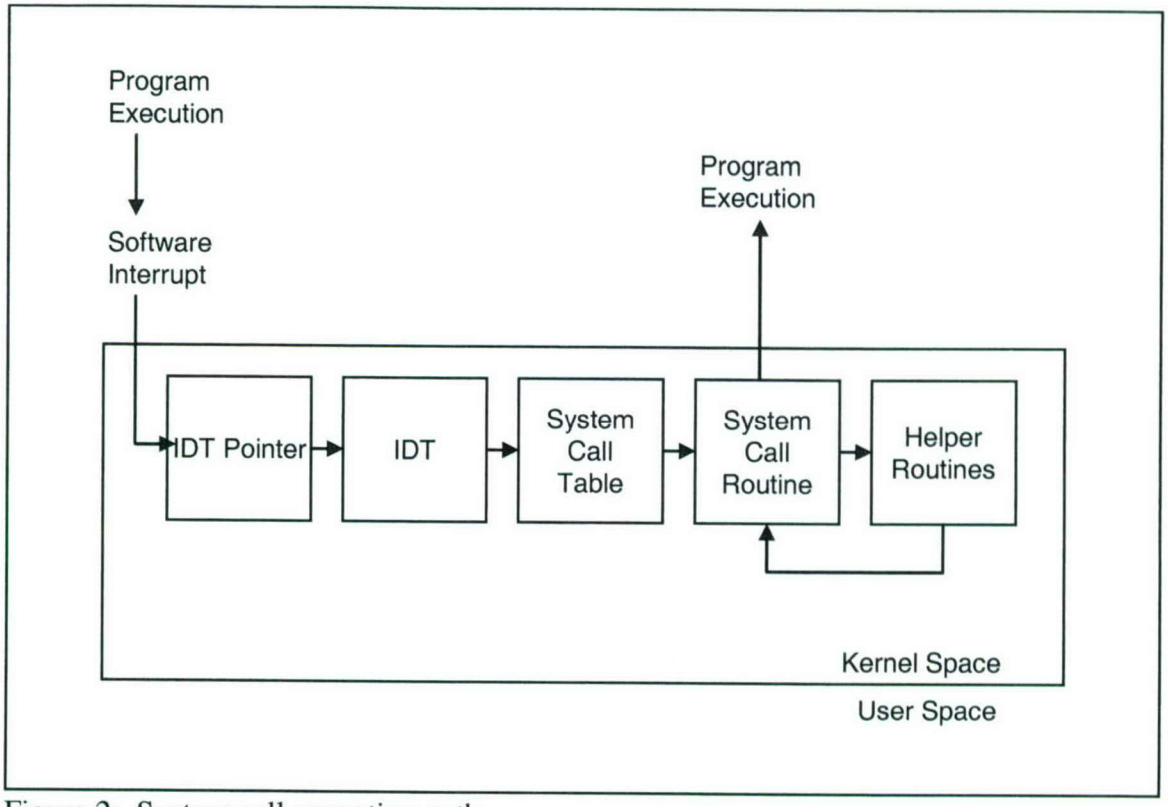

Figure 2: System call execution path

This layered architecture and modular design of the Linux operating system allows individual components to be modified without affecting the rest of the system. This is a desirable trait from a software engineering perspective. However, it also simplifies the creation of deceptive interpreters. They can target a single component and have the modified results propagate to subsequent layers.

5. Taxonomy of Deceptive Interpretation in Linux

This section presents a defense-centric and functional taxonomy of software deceptive interpretation on the Linux operating system. Analysis is performed from the point of view of the host system to enumerate all possible ways through which deceptive interpretation can take place. Categorization of deceptive interpretation is based on the mechanisms that deceptive interpreters employ. For most categories, the mechanisms will directly correlate with the host system components that need to be modified in order for the deceptive interpreter to function. In these cases, the goal of the taxonomy is to identify common methods for detecting deceptive interpreters in the same category. For other categories however, the mechanism are abstract and do not directly lead to an obvious method of detection. The following is a graphical overview of the taxonomy.

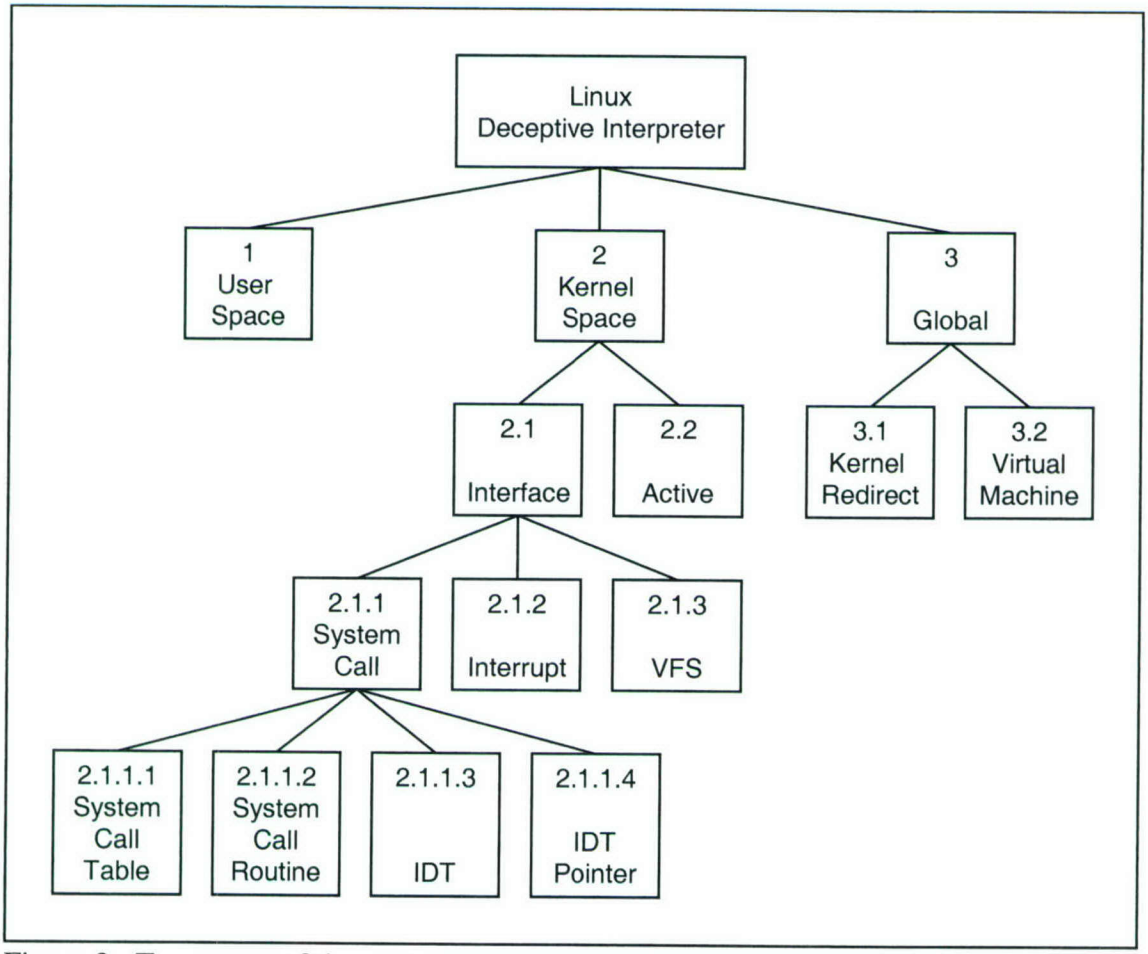

Figure 3: Taxonomy of deceptive interpretation in the Linux operating system

The goal of the taxonomy is to serve as a basis for developing tools that are able to provide categorical protection against deceptive interpretation and rootkits. *Categorical protection* refers to the ability of detection methods to work against entire categories of deceptive interpretation under stated assumptions. A category-specific detector should be able to detect all deceptive interpreters that fall into a particular category regardless of the

implementation details of the actual rootkits. It should also work against new rootkits never before seen. This is hardly possible to achieve through signature-based approaches. The assumptions made by the detector must be reasonable and deemed acceptable by the operator of the system. Steps should be taken to ensure the assumptions are indeed valid.

User space deceptive interpreters target components that reside in user space. *Kernel space deceptive interpreters* target components in kernel space. *Global deceptive interpreters* affect components in both user and kernel space. The following sections will describe each category of deceptive interpretation in more detail.

5.1. User Space Deceptive Interpreters

Deceptive interpretation can be performed in user mode by statically modifying the binary and data files for applications and libraries. Trojan files would remain functional in appearance, but they are able to return incorrect results according to the policy of the deceptive interpreter. For example, a Trojan horse `ls` binary file could be programmed to never display files belonging to the rootkit. System administrators using this command to check their system will not be able to detect those files. Data files could also be used to perform deceptive interpretation. For example, the mapping between hostnames and IP addresses are stored in a data file. Modification of that file will allow the deceptive interpreter to redirect network traffic for operations that refer to hostnames.

User space deceptive interpreters are relatively unsophisticated and have been more commonly used in early rootkits. The coverage is rather narrow and limited to only those operations that directly rely on the modified files. However, most people take commands like `ls` for granted. A properly constructed user space deceptive interpreter would appear to be completely normal to the user and would not arouse suspicion. Special file integrity checkers such as *Tripwire* [16] and *AIDE* [17] are necessary for detection of this type of deceptive interpretation.

It is also important to note that Trojan horse files do not necessarily have to physically replace the original files. The deceptive interpreter could simply change the environmental variables so that execution is redirected to the Trojan horse file at the system level [18, 19]. For example, the path definitions could be modified so that when the user types `ls` on the command prompt, an entirely different `ls` command is executed. In this case, the original binary would remain unmodified and file integrity checkers would not detect the deceptive interpreter. It is not only necessary to verify the integrity of the files themselves, but also the integrity of everything they depend on, such as environment variables, system configuration, and `shell` functionalities.

5.2. Kernel Space Deceptive Interpreters

Operations in the kernel mode function with full privilege including direct access to hardware and memory. As a result, deceptive interpreters residing in kernel space are

much more powerful. They have the potential to completely control the affected system by mediating all system services. Furthermore, access restrictions do not apply to operations in kernel space. User processes are isolated from each other through various protection mechanisms. However, the kernel is assumed to be trusted. There are no mechanisms protecting processes in either user or kernel space from malicious intervention by a component of the kernel.

A kernel space rootkits must first find ways to migrate to kernel space in order to operate. One way to do this is to take advantage of the loadable kernel module (LKM) feature of the Linux operating system. This feature is used for adding modules such as device drivers into the kernel at runtime without recompiling. However, having the LKM feature also makes rootkit injection much easier. Rootkits can simply be loaded as a kernel module enabling them to gain entry into kernel space. Another way to inject rootkits into the kernel is to write directly to memory. The `/dev/kmem` interface is a special device that can provide direct access to the memory including regions occupied by the running kernel. Users or programs with root privilege can perform write operations on it. Deceptive interpreters can alter the kernel by overwriting the kernel image in memory through this interface. These two methods utilize features provided by the Linux operating system. They are the easiest and by far the most common methods used by rootkits to enter kernel space. Other methods include modification of the kernel image on disk. Rootkit code can be injected into the disk image of the kernel and can become operational after the system reboots. Exploits can also target kernel code directly. Buffer overflow attacks against parts of the kernel such as device drivers will result in execution of malicious code in kernel space.

Kernel space deceptive interpreters could be divided into two categories. *Interface deceptive interpreters* make up the first category. They operate by intercepting and hijacking functions associated with interfaces. They are passive in nature. Once installed, their malicious code must wait to be invoked by some other component of the system. *Active deceptive interpreters* make up the second category. They do not rely on interfaces but rather actively monitor the system to perform deceptive interpretation.

5.2.1. Interface Deceptive Interpreters

The Linux operating system contains several layers of abstraction and system interfaces. *Interface deceptive interpreters* operate by hooking interface functions and redirecting them to execute malicious code. The most important interface is the system call interface between user and kernel space. As a result, the vast majority of deceptive interpreters in published rootkits are system call deceptive interpreters. However, other system interfaces do exist and could also become targets of deceptive interpretation. Two of these other interfaces are the interrupts interface and the virtual file system interface.

5.2.1.1. System Call Deceptive Interpreters

Deceptive interpretation can be performed anywhere along the execution path. A rootkit could convince programs that one plus one equals three. Such computations reside exclusively in user space. Therefore, the deceptive interpreter for them would also be in user space. However, rootkits are more interested in hiding files, processes and network connections. Operations relating to these objects require kernel services that user space applications must obtain through the system call interface. Hijack of system call functions would allow the deceptive interpreter to mediate all user space access to those objects. Consequently, deceptive interpreters in most modern rootkits are system call deceptive interpreters.

There are four places where system call deceptive interpreters can place a hook for intercepting system calls.

5.2.1.1.1. System Call Table

The system call table is a linear array of pointers to system call routines. It basically serves to map system call numbers to the routine entry points in memory. Modification of this table will change the mapping and enable rootkits to point specific system calls to their own Trojan system call routines.

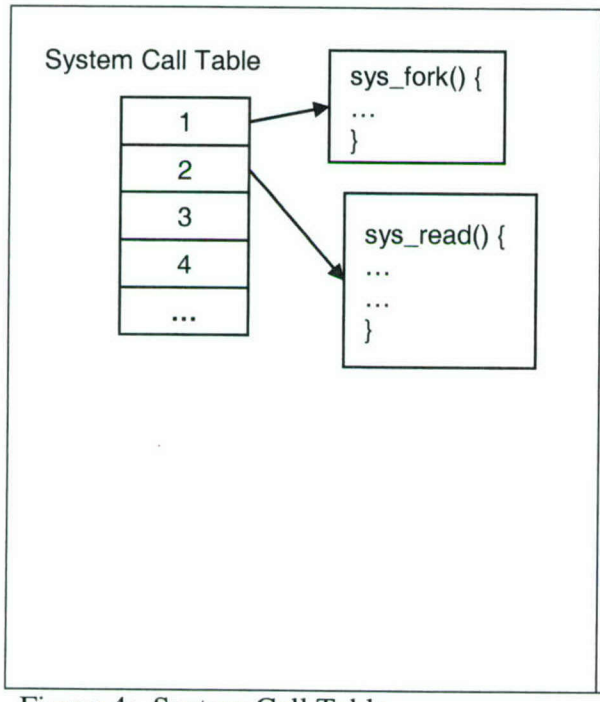

Figure 4: System Call Table

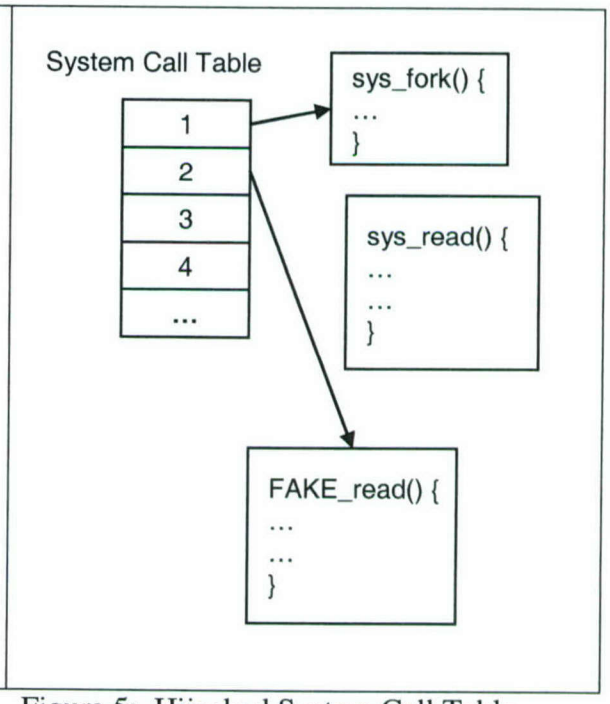

Figure 5: Hijacked System Call Table

This is the earliest method published for intercepting system calls described in 1997 [20]. Detection of this deceptive interpreter requires checking the integrity of the system call table. The system call table is a static data structure. It should not change during normal operations of the system and any modification would indicate the presence of a rootkit.

5.2.1.1.2. System Call Routines

Deceptive interpreters can also tamper with the system call routines themselves. This could involve overwriting the routine code segment in memory with a Trojan version, or overwriting only a portion of the routine to perform a jump to a Trojan routine that performs deceptive interpretation.

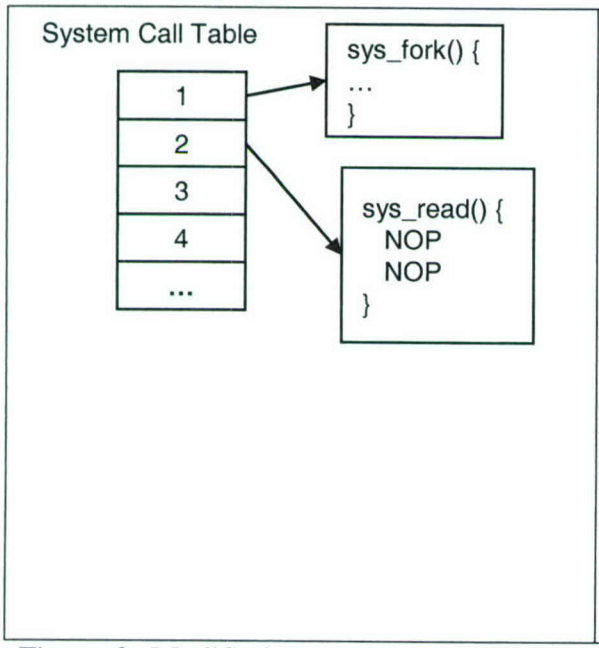

Figure 6: Modified system call routine

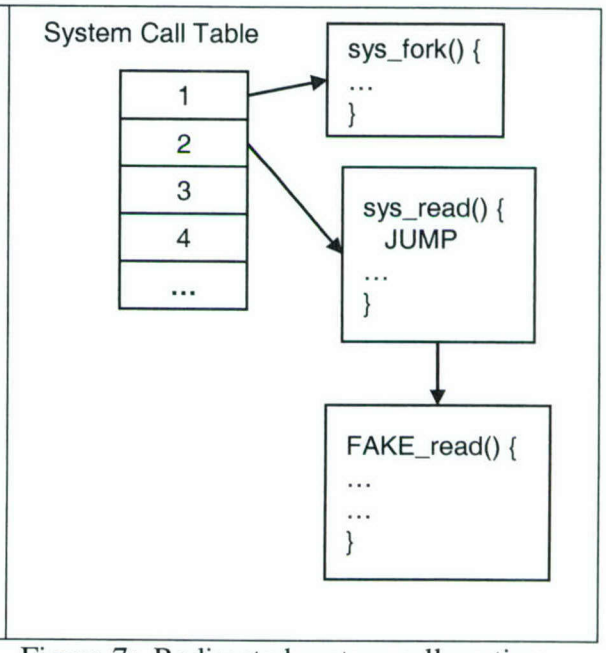

Figure 7: Redirected system call routine

The latter method was published in 1998 [21-23]. The main advantage of this method is that it is much stealthier. The system call table is not modified so it cannot be detected by system call table integrity checks. Detection of this type of deceptive interpreter requires verification of all critical system call routines. The published implementation places the jump at the beginning of the routine. However, it is theoretically possible to perform the jump anywhere from the beginning of the routine to right before the return statement. Therefore, it is necessary to verify the integrity of the entire routine code body. This is much harder than simply verifying the integrity of the system call table with a length of only 1024 bytes. Another problem brought forth by this method is that system call routines often call other kernel functions in the course of their execution. In order to completely ensure that a system call has not been redirected, the entire tree encompassing all possible execution paths must be verified. In an experiment tracing through system call execution paths, it was shown that an overwhelming majority of the more than 2000 exported functions in the Linux kernel are either directly or indirectly called by system call routines. Verification of all the code that could potentially execute in the course of performing system calls is not a trivial endeavor. It would be a simpler solution to just verify the integrity of the all kernel code segments in memory.

5.2.1.1.3. Interrupt Descriptor Table

The IDT is an array of interrupt descriptors that contains pointers to interrupt handling routines. Interrupt 0x80 is the software interrupt used for system calls. Modification of this entry could redirect system calls to a Trojan system call table which could in turn point to Trojan system call routines.

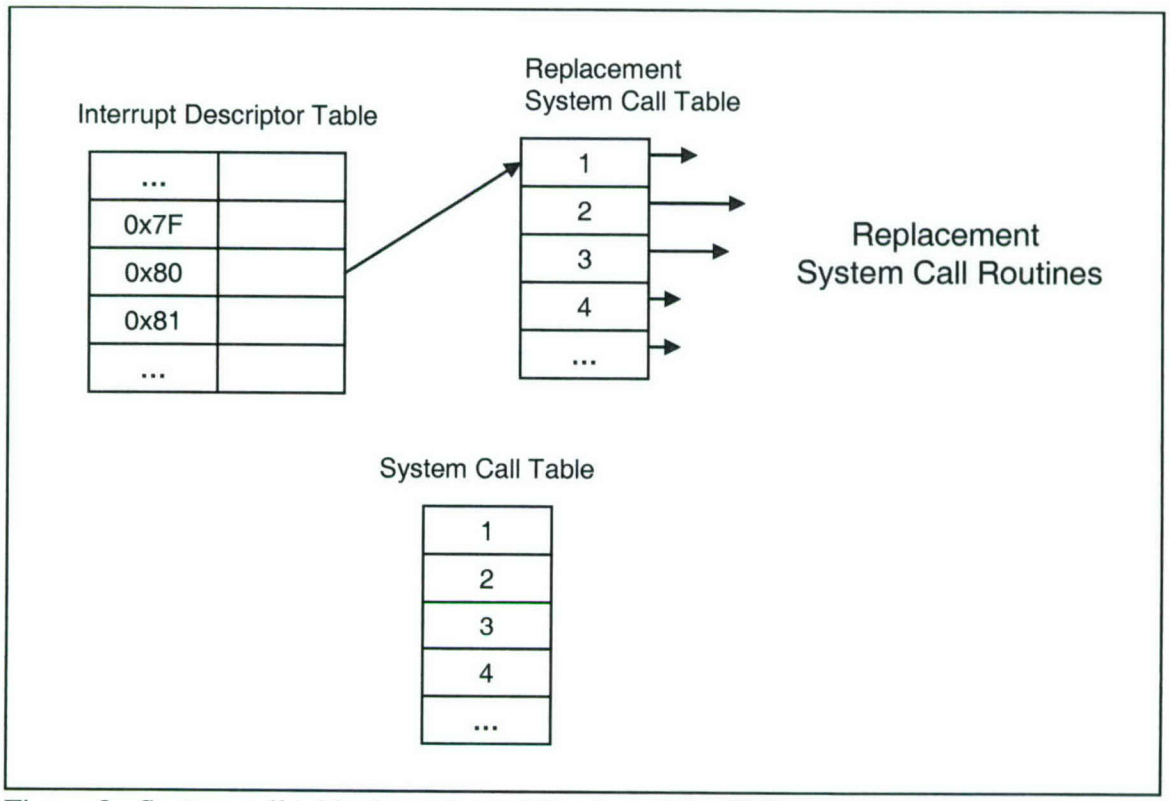

Figure 8: System call table through modification of the IDT

This method was first described in 2002 [24]. It involves the creation of a new system call table. The original system call table and system call routines would remain intact without modification and pass scans by system integrity checkers. Detection of the deceptive interpreter requires looking specifically at the IDT at interrupt 0x80 to see that a different system call table is being used. This is not hard to do. However, many of the rootkit detectors at that time and even today still do not perform this check.

5.2.1.1.4. Interrupt Descriptor Table Pointer

The initial stages of interrupt handling are performed exclusively by hardware. The IDT marks the entry into the software realm. For most system architectures, the location of

the IDT is stored in a CPU register. It is possible to change the content of that register to point to an entirely different IDT.

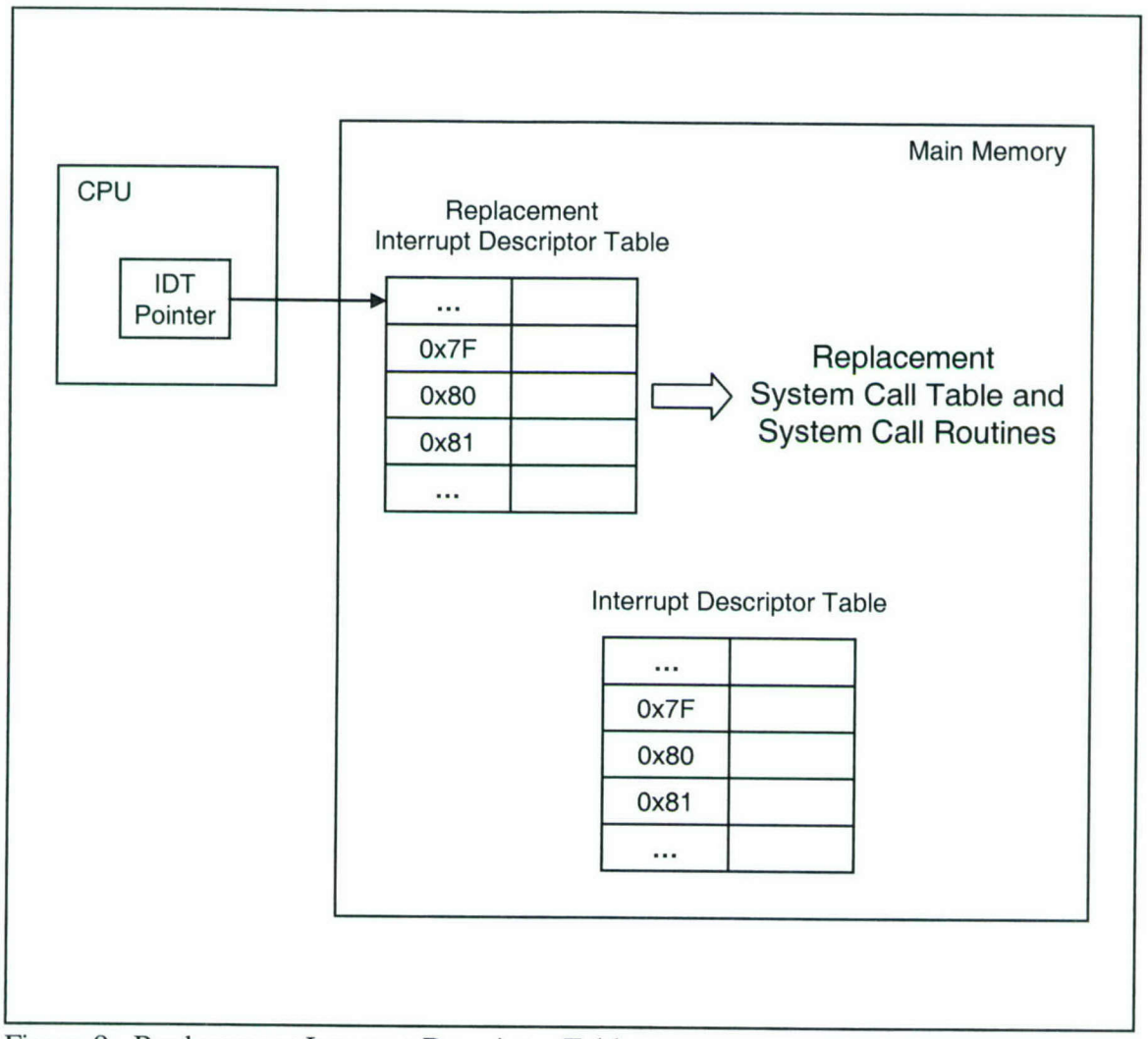

Figure 9: Replacement Interrupt Descriptor Table

This method of performing deceptive interpretation has not yet been published. A prototype implementation was created as part of this project. It involved creating duplicates for the IDT as well as the system call table. This method would not have a signature in memory or secondary storage whatsoever. The original data structures and routines are untouched and scans aimed at them would not reveal the presence of the deceptive interpreter. The interrupt descriptor and system call tables do not have any specific format or header requirements that could be used as a signature to specifically identify them. The duplicate structures could be easily disguised [25]. The only way to defeat this deceptive interpreter is to query the CPU itself for the location of the IDT to ensure the correct one is being used. The actual implementation of this check is architecture specific. However, all interrupt driven architectures store the IDT pointer in CPU registers.

5.2.1.1.5. System Call Deceptive Interpreters Summary

A shortcoming of system call deceptive interpreters in general is that their coverage is limited to user space. Code executing in kernel space can call kernel functions directly without having to go through the system call table. As a result, kernel space operations are inherently outside the coverage of system call deceptive interpreters. In reality, this is not a serious shortcoming. The targets of deceptive interpreters for the most part lie in user space. However, hijacking general kernel functions instead of specific system call functions could easily extend the coverage of a deceptive interpreter into kernel space. Furthermore, some kernel functions may serve as nice bottle-necks for certain functionalities. The deceptive interpreter could hijack a single kernel function instead of multiple system call routines and thus reduce its signature.

5.2.1.2. Interrupt Interface Deceptive Interpreters

The interrupt interface serves as the sole entry point into the kernel. All software interrupts as well as hardware interrupts are first processed by this interface. The potential coverage of interrupt interface deceptive interpreters is greater than that of system call deceptive interpreters.

No implementations of this mechanism have been published thus far. System call deceptive interpreters described previously in sections 5.2.1.1.3 and 5.2.1.1.4 that tamper with the IDT are technically interrupt interface deceptive interpreters. They were placed in the system call category of deceptive interpreters because their mechanism of operation is specifically aimed at the system call execution path. Deceptive interpretation that specifically targets the interrupt interface is more difficult because it operates at a very low level. One potential attack is to tamper with the timer interrupt handler and the operating system scheduler. Trojan schedulers can pretend to run certain processes without actually doing so. Services such as intrusion detection systems and anti-virus programs could be victims of denial of service attacks that are very stealthy. The processes would still show up in the process list and appear to have CPU time, but their code would never actually execute. The interrupt interface also mediates communication with interrupt driven devices. For example, keyboard loggers and network sniffers can be placed at the interrupt handler level and become very stealthy.

The detection of this category of deceptive interpretation involves verification of the interrupt handling execution path. Integrity checks need to be performed on the IDT pointer, the IDT, the interrupt handling routines, and helper routines called by the interrupt handling routines.

5.2.1.3. Virtual File System Deceptive Interpreters

Another layer of abstraction is the virtual file system that provides a common software interface for the underlying physical file system. Deceptive interpretations at the virtual file system interface can alter the user space view of the file system. This is especially useful for enforcing storage related policies such as hiding or protecting files.

The top level of the virtual file system interface actually resides on the system call interface. It implements system call routines such as `open`, `stat`, and `chmod` for performing file operations. These routines in turn call the specific implementation routines for the underlying file system. Deceptive interpretation can be done either at the top level system call routines or at the implementation routines. It is possible to perform deceptive interpretation on the virtual file system without modification to the top level system call routines, but it is not possible to do so without affecting the system call execution path. In this sense, the virtual file system is a subset of the system call interface. Methods used to detect system call deceptive interpreters would also detect most, but not all, virtual file system deceptive interpreters. The virtual file system involves another layer of redirection that is analogous to the system call table on the system call interface. Inodes associated with each file contain pointers to routines that implement various file operations. Deceptive interpretation can be performed by modification of those pointers to point to Trojan routines. This mechanism does not need to modify any objects in the kernel. Only inodes, which reside in user space, need to be modified.

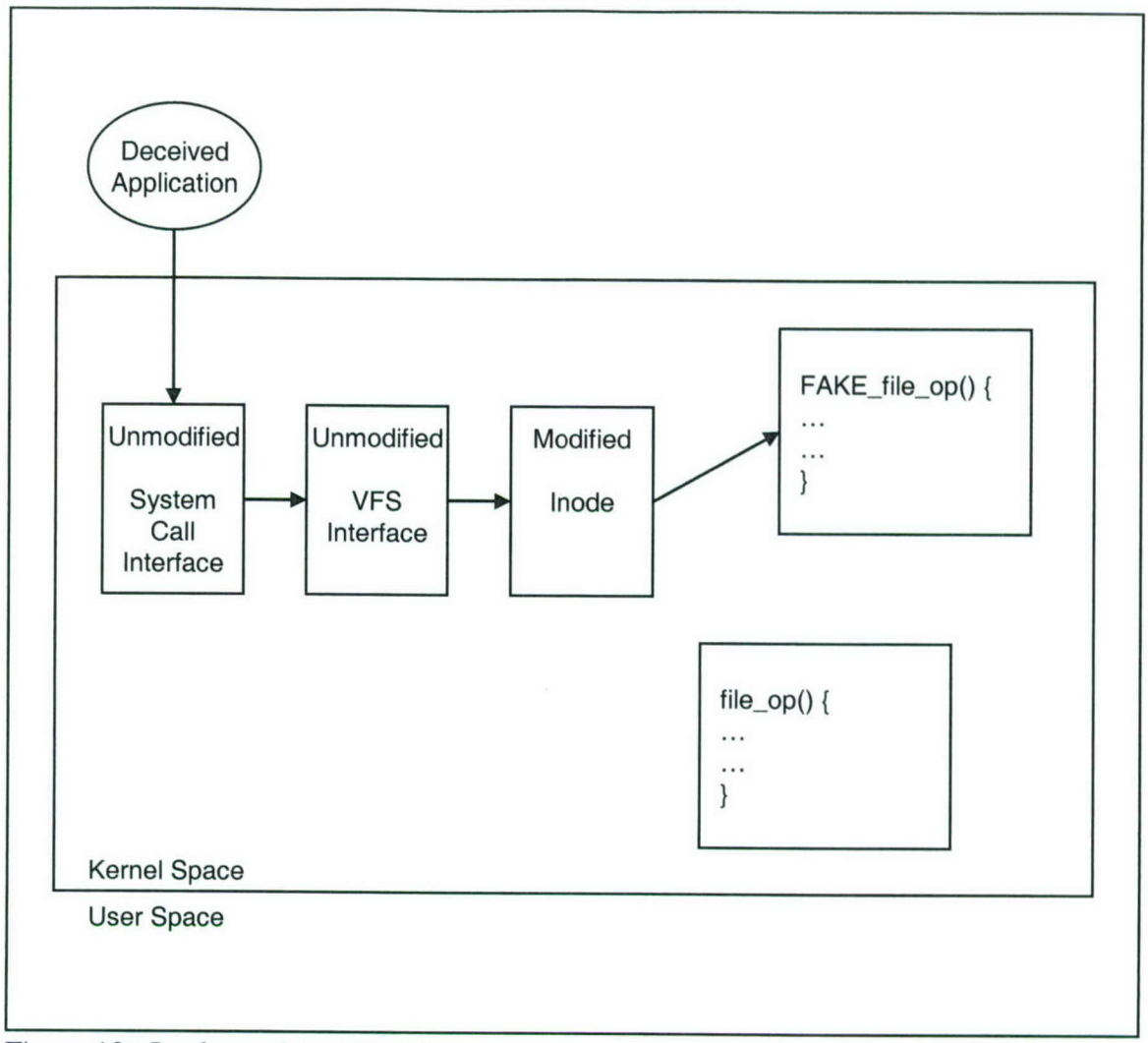

Figure 10: Inode routine redirection

A component of the virtual file system that is commonly targeted by deceptive interpretation is the *proc file system*. Files within the proc file system do not physically exist in storage, but rather are dynamically generated when the file is accessed. These files provide information to user space processes about hardware status, kernel configuration, and system operations. This interface is accessed through the `/proc` and `/dev` directories. Different aspects of the system are further divided into subdirectories such as `/proc/interrupts` and `/proc/sys`. Many system audit utilities such as `top` and `netstat` rely on information provided through this interface. Deceptive interpretation at this layer could provide false information to those utilities and undermine their results.

Deceptive interpretation using the `proc` file system was first published in 2001 [26]. The implementation works by hijacking kernel functions that generate the virtual files. It is able to hide files, processes, and network connections from system utilities that rely on the `/proc` directory. The article claims that this method does not affect system calls, but

it is not completely true. User space processes access the file system, including the virtual file system, through the system call interface. The system call routines themselves, the top layer of the system call interface, are not modified, but the functions generating the virtual files are called by the system call routine. Virtual file system deceptive interpreters invariably lay on the system call interface execution path, but they operate on a distinct layer of abstraction. The coverage of deceptive interpreters that operate exclusively through the virtual file system is limited. However, the nature of their coverage makes them a threat. Human system administrators would often browse through the `/proc` and `/dev` directories to check for inconsistencies. Some rootkit detection tools also perform sanity checks between information retrieved through system calls and the virtual file system. Virtual file system deceptive interpreters could be used in conjunction with other techniques to enhance the coverage of the overall deceptive interpreter.

Detection of virtual file system deceptive interpreters requires checking the integrity of the routines on the virtual file system interface as well as helper routines called by those routines. Furthermore, inode information on key files and directories, such as those in the proc file system, also need to be verified to ensure routines for file operations are not redirected.

5.2.1.4. Interface Deceptive Interpreter Summary

Deceptive interpretation invariably involves changing the data or code actually executed by the system. Interface deceptive interpreters intercept system commands issued by applications during execution by tampering with the underlying system interfaces. As a result, they are considered passive. Their malicious code gets executed only when those functions are called by another component. Most of the time, they remain dormant in the system. This mode of operations has two major implications. First, interface deceptive interpreters are not fully capable of keeping track of the current state of the system. They are only executed when their particular interface is invoked. Second, the signature of passive deceptive interpreters is typically larger than necessary. Their malicious code segments are dormant most of the time, but nonetheless they need to be present in the system. Checking the integrity of the system interfaces would invariably reveal the presence of an interface deceptive interpreter.

5.2.2. Active Deceptive Interpreters

Active deceptive interpreters are defined by mechanisms of operation that do not rely on interception of interface function calls at run time. Instead, they tamper with the application data and code segments themselves. As a result, integrity checks performed on interfaces would not reveal the presence of active deceptive interpreters.

Active deceptive interpreters could tamper with applications in a number of ways. A systematic analysis of these methods with sample implementations was published in 2002 [27]. The paper describes several events during initialization of a process prior to execution at which redirection or replacement of application code can occur. Execution of user space processes is handled by a system call that in turn calls helper routines that load the binary into memory, handle the binary format, and perform dynamic linking during initialization. The particular implementations presented in the paper used the system call interface to intercept those events. As a result, these implementations would be detected by integrity checks for the system call interface. Indeed, the easiest way to perform active deceptive interpreters involves hijacking system interface functions. However, the function hijackings they perform are only part of the event detection process. Their actual mechanism of operation involves replacement of application code. The function hijacking at an interface is not absolutely necessary and cannot be used as the basis for detecting this category of deceptive interpretation. The paper pointed out alternative implementations that do not make any modifications to system interfaces. Active deceptive interpreters could simply poll for specific events and respond to them. For example, a rootkit can continuously scan the process list and temporarily remove its files if the start of a *Tripwire* process is detected. This method is not implemented in published rootkits. However, it is important to realize they are possible and account for them in research concerning deceptive interpretation.

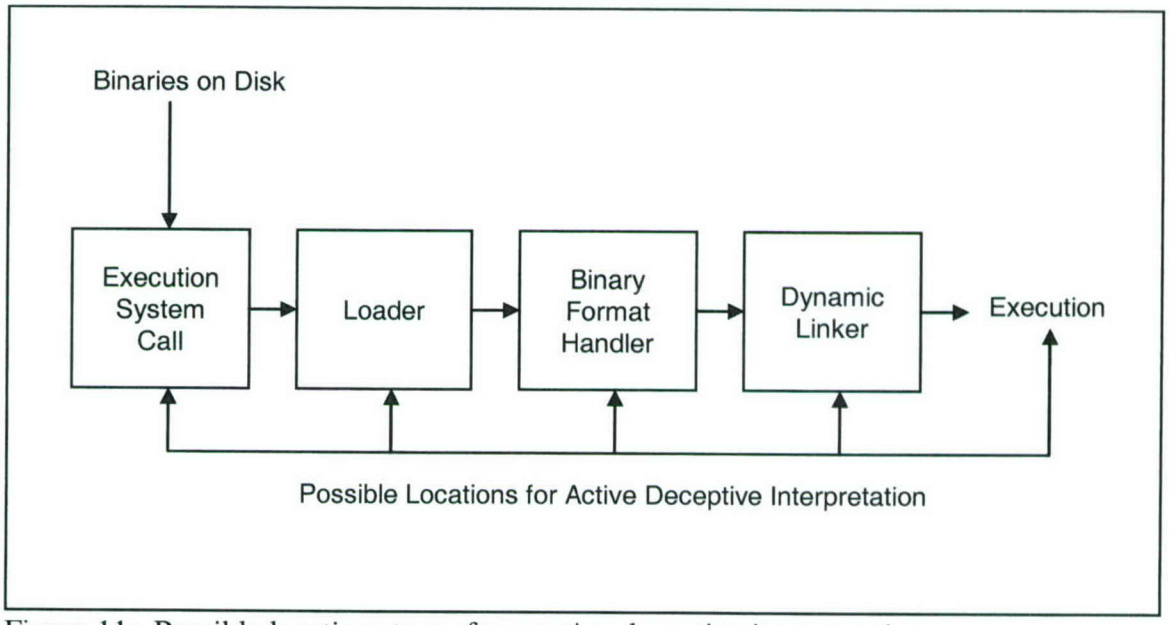

Figure 11: Possible locations to perform active deceptive interpretation

Active deceptive interpreters are indeed very hard to deal with. First, it is no longer sufficient to simply verify the integrity of system interfaces. In order to detect this type of deceptive interpretation, all code segments that run on the system need to be verified at execution time. Furthermore, deceptive interpretation could be performed by tampering with application and operating system data. Integrity checks must also be performed on relevant volatile data in both kernel and user space at run time. This would require the

checkers to constantly keep track of the state of the system and perform verification whenever data is accessed. These methods are computationally expensive, hard to implement, and completely impractical in most systems. Furthermore, it brings up the inherent problem of running checks on an untrustworthy system. The application that is supposed to check the integrity of code segments could itself be compromised. With interface deceptive interpreters, it is always possible to bypass the particular interface and access resources directly to perform integrity checks. For example, processes running in kernel space can bypass the system call interface and check the integrity of the system by scanning memory directly. There is no simple way to bypass an active deceptive interpreter. Under the hardest attacker assumption, the deceptive interpreter would have complete knowledge of the system and the protection schemes being employed. It would know how to scan for threatening code segments and neutralize them. As long as the attackers know how the protection schemes operate, they should be able to write active deceptive interpreters that evade detection.

Active deceptive interpreters are especially dangerous when they are used in conjunction with interface deceptive interpreters. Active deceptive interpreters are very effective in defeating specific countermeasures. Interface deceptive interpreters, on the other hand, are very complete in their coverage. Their characteristics nicely complement each other. For example, the active deceptive interpreters could defeat specific advanced scanners while interface deceptive interpreters could defeat more general threats such as a system administrator browsing through the system. Under the hardest attacker assumption, the potential coverage of the combined deceptive interpreter could be complete against software based means of detection.

5.3. Global Deceptive Interpreters

Global deceptive interpreters perform deceptive interpretation through complete system emulation by creating a comprehensive wrapper around processes or even the operating system. This could be done either by redirecting execution to a malicious copy of the kernel, or by deployment of a virtual machine on the system.

5.3.1. Kernel Redirection Deceptive Interpreter

The interrupt interface is the lowest level interface on Linux systems serving as the sole entry point into the kernel. All kernel operations, including system calls and hardware interrupts, must go through this interface. The IDT pointer stored in registers on the CPU serves as the single entry point to the interrupt interface and the kernel. Modification of that pointer can result in redirection of the entire kernel.

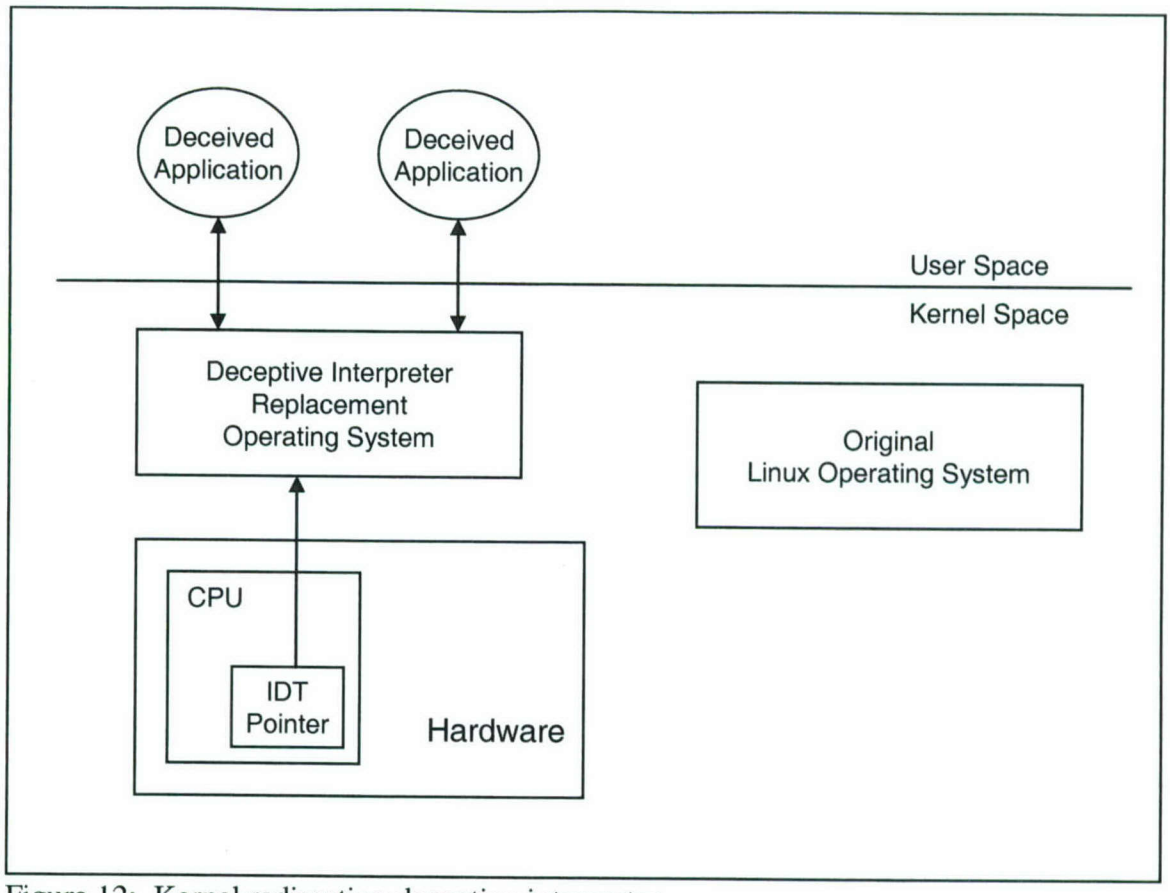

Figure 12: Kernel redirection deceptive interpreter

The attacker can allocate high memory in either kernel space or user space, replicate the entire operating system, modify it for deceptive interpretation, and redirect kernel execution to the replicate. The advantage of such an approach is that the original kernel stays intact. Scans that check the integrity of the kernel or look for attack signatures will perform their operations on the intact original kernel. However, actual execution will be performed by the malicious copy of the kernel.

The problem with this approach is that for most architecture, including Intel x86, checking the value of the IDT pointer is relatively trivial. The processor instruction for doing so is available in user space. Applications can check the value of the IDT pointer and realize an entirely different kernel is running on the system. Since applications do not need to go through the system call interface to perform the check, there is no simple way for a deceptive interpreter to intercept the operation to return a false result. Any application that knows what to check for can detect the redirection of the kernel. This problem can be resolved by deploying a virtual machine deceptive interpreter.

5.3.2. Virtual Machine Deceptive Interpreter

A great deal of research effort has gone into the development of honey pots and virtual machines. These systems are capable of emulating complete environments. Processes running inside them theoretically cannot access the underlying native operating system other than through the virtual machine. It is important to realize that this body of research could be used maliciously as deceptive interpreters. The same methodologies used in virtual machines and honey pots to fool attackers into thinking they are in a successfully compromised machine could be applied to fool legitimate users and processes into thinking they are in an uncompromised machine.

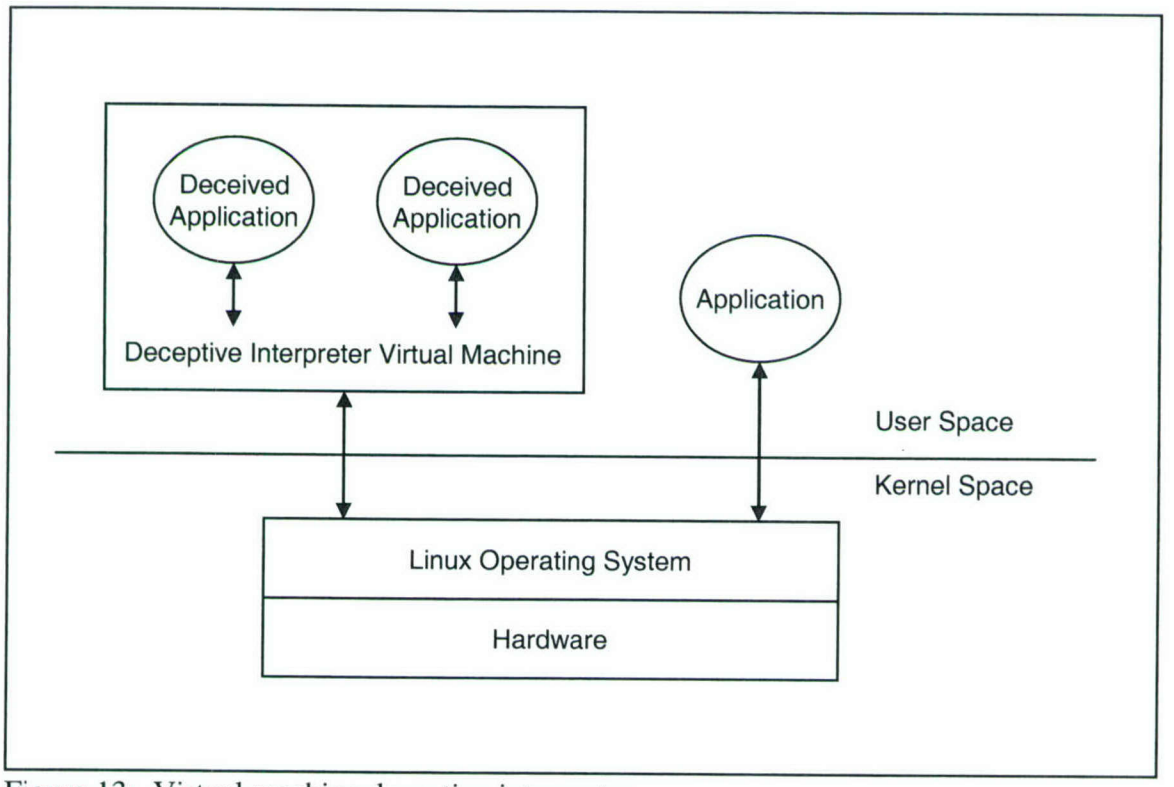

Figure 13: Virtual machine deceptive interpreter

Deceptive interpreters in currently published rootkits only tamper with a relative handful of system functionalities. The scale of virtual machines is several orders of magnitude larger than that of most rootkits. However, many tools already exist that perform hardware and operating system emulation. Many of these tools are open-source and widely available [28]. Attackers could easily modify these tools and apply them in the context of deceptive interpretation.

There are several potential advantages to the virtual machine approach for deceptive interpretation. First, a large body of published research already exists for honey pots and virtual machines. Many of the technical problems have already been resolved by legitimate researchers. Attackers can take finished products and with only a few modifications, convert them into very effective deceptive interpreters. Second, virtual machine deceptive interpretation is extremely powerful. They could mediate every single

operation in the system to enforce their policy globally and could not be bypassed. The coverage could potentially be complete against software means of detection even without the hardest attacker assumption. Active deceptive interpreters require knowledge of exactly what countermeasures are deployed against it to achieve complete coverage. Virtual machine deceptive interpreters could conceivably achieve complete coverage without the need for complete knowledge of the system. It should be possible to construct a global deceptive interpreter capable of defeating all software based means of detection in a generic system.

By the same token, many of the problems faced by virtual deceptive interpreters are the same as those faced by honey pots and virtual machines. The most significant problem is that virtual machines tend to be very inefficient because every single operation needs to be interpreted. Installing a virtual machine deceptive interpreter on a machine would significantly degrade performance, which could lead to detection. It would require either a human user noticing the performance degradation, or a process on a separate computer using an independent clock. Another problem is that virtual machines tend to be very large and complex. Transferring them to a foreign host, compiling, and then deploying them is not an easy process.

6. Overview of Existing Rootkits

Many of the mechanisms included in the taxonomy are not actually employed in a published rootkit. They represent potential mechanisms from the host system point of view that could be used by future rootkits to perform deceptive interpretation. In practice, a vast majority of rootkits employ user space or system call deceptive interpreters. The following figure provides a graphical view of categories of the deceptive interpreters actually implemented in published rootkits.

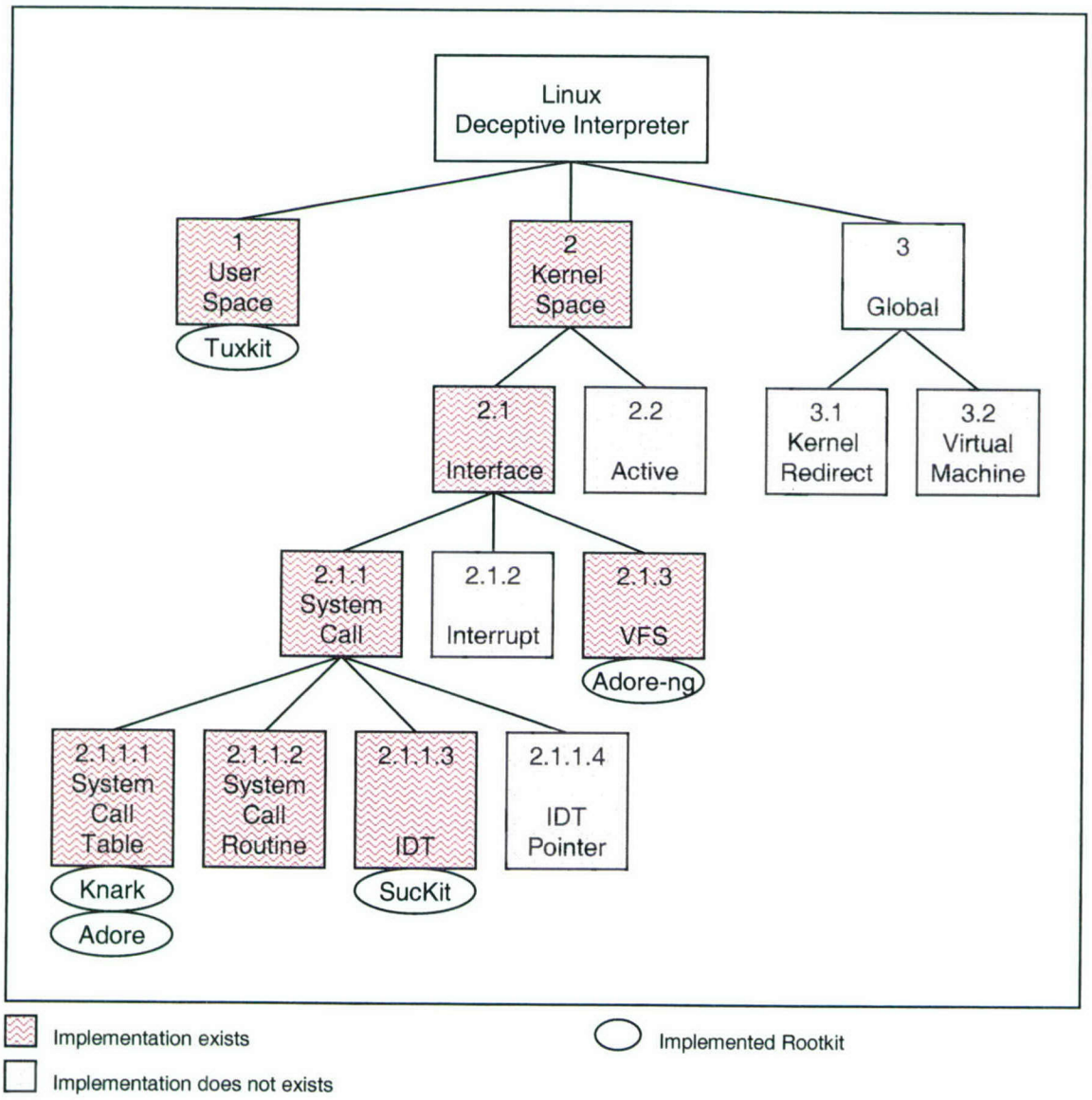

Figure 14: Summary of existing rootkits

The following sections will present brief summaries of stereotypical published rootkits in several categories. These rootkits are available for download in one of the many rootkit repositories on the Internet [2, 8].

6.1. TuxKit – User Space Deceptive Interpreter.

TuxKit was written by a Dutch group called Tuxtendo and published in May 2001 [29]. It contains six Trojan binary files, a read-me file, and an installation script. The deceptive interpreter consists of Trojan binaries for basic diagnostic functions used by the system admin including `ls`, `df`, `netstat`, `find`, `locate`, and the `libproc` library. The Trojans are able to hide files and processes. Other core components of *TuxKit* include a network traffic sniffer and a ssh Trojan with sshd-config capability that provides a backdoor into the system.

Installation of *TuxKit* is easy and straightforward. The setup script only requires users to provide a password for subsequent reentry and specify the listening ports for `sshd` and `bnc`. The installation of the deceptive interpreter involves first killing `syslogd` to prevent its activities being logged, then the files are extracted to a directory named tux under /dev. This directory is used to store hidden files such as `.addr`, `.cron` and `.proc` that are to perform deceptive interpretation.

Detecting *TuxKit* is fairly simple. As a standard security practice, the system administrator should deploy a file integrity checker such as *Tripwire* or *AIDE* on their system. All system files and libraries should be protected including ones modified by this rootkit. Furthermore, searching for the existence of `/dev/tux`, `/usr/bin/xsf` and `/usr/bin/xchk` directories can also reveal the presence of the rootkit.

6.2. Knark – System Call Table Deceptive Interpreters

The first released version of *Knark* was published in June 1999 by Creed [30, 31]. *Knark* is loaded as a LKM after the system has been rooted. It hides its presence using the *modhide* tool included in the rootkit distribution. Deceptive interpretation is performed by redirection of seven system calls by modifying the system call table:

- fork
- read
- execve
- kill
- ioctl
- settimeofday
- clone

The most current version, *Knark-0.59*, can hide files and directories, conceal TCP and UDP connections, hide running processes, change UID/GID of a running process, and allow unauthenticated privileged remote execution. Furthermore, it is distributed with a number of precompiled exploits for gaining control of a system to install the rootkit. They include the Lpdx exploit for the line printer service in Red Hat boxes, the T666

exploit for BIND 8.2.1 and the Wugod exploit for the Washington University ftpd 2.6.0 daemon.

Knark is feature-rich, sophisticated and hard to detect. Checking the integrity of system binaries using file integrity checkers such as Tripwire or AIDE will not reveal its presence. Deceptive interpretation occurs inside the kernel leaving application and library binaries untouched. Detection of *Knark* requires specialized system security utilities that can check the integrity of the system call table.

6.3. Adore – System Call Table Deceptive Interpreters

The first version of *Adore* was made public in 1999 by Stealth [32, 33]. Like *Knark*, it is injected into the kernel as a loadable kernel module and performs deceptive interpretation by intercepting system calls through modification of the system call table. The most current version, *Adore-0.53*, alters twelve system calls:

- kill
- write
- fork
- clone
- close
- symlink
- mkdir
- stat
- lstat
- open
- oldstat
- oldlstat
- getdents

The configuration script of *Adore* makes its installation easy and flexible. Besides using the default configuration, the makefile can be edited to customize the installation. Two binary executables, `ava` and `startadore`, are created after a make. `Startadore` is used to insert and remove the *Adore* modules from the kernel. `Ava` is a user space program that can be used to control how *Adore* performs deceptive interpretation.

Similar to *Knark*, *Adore* hides processes, ports, files and directories. It also provides a root shell backdoor and has the ability to remove PID's permanently. However, *Adore* does not implement any remote access features like *Knark*. Instead, it has a smart PROMISC flag hiding feature that can be used to conceal network sniffers.

6.4. SucKIT – Interrupt Descriptor Table Deceptive Interpreters

SucKIT was first introduced in 2001 by sd and devik in 2001 [34]. It is designed to be used on a Linux system without utilizing the LKM feature. Instead, it is injected into the

kernel by overwriting the memory image of the running kernel through the `/dev/kmem` interface. *SucKIT* works by changing the IDT at interrupt 0x80 to point to a Trojaned replacement system call table. The original systems call table remains intact and is simply not used. The following eighteen system calls are affected:

- olduname
- fork
- clone
- open
- close
- read
- kill
- getdents
- getdents64
- ioctl
- stat
- fstat
- mmap
- munmap
- getpid
- readdir
- readlink
- lseek

SucKit can hide specific process, files and network connections. It provides a password protected remote access connect-back shell that is able to bypass most firewall configurations. Its footprint is limited to memory and it does not make any changes on the file system. The rootkit is also completely self-contained without the need for any outside libraries on the host machine. However, *SucKit* is not portable and only works with i386-linux specific systems.

6.5. Adore-ng – Virtual File System Deceptive Interpreters

Adore-ng was created in 2004 by Stealth, who also created Adore [35]. It is loaded as an LKM and has similar features and capabilities as the original Adore. The difference is that instead of redirecting system calls directly, *Adore-ng* is placed deeper in the kernel on the virtual file system layer of the operating system. Leaving the system call interface intact, it has a smaller signature and footprint on the system, making it harder to detect. *Adore-ng* also takes control of the `/proc` file system. Many user space programs, especially system audit utilities and host-based intrusion detection systems, rely on this interface for information. Deceptive interpretation in the `/proc` file system will undermine the results of those programs.

The most current version, *Adore-ng-0.32*, is a sophisticated full featured rootkit with the following capabilities:

- file and directory hiding

- process hiding
- socket hiding
- full featured backdoor
- syslog filtering
- wtmp/utmp/lastlog filtering
- optional re-linking of LKM to allow persistence across reboots [36]

Adore-ng can perform deceptive interpretation with the help of `ava`. It also provides a separate set of commands that can be used directly from the command line. *Adore-ng* currently is one of the most powerful and stealthy rootkits available to the public. It incorporates many of the most advanced features in rootkits such as a sophisticated backdoor and the ability to persist across reboots. The modifications it makes for deceptive interpretation are buried deep inside the kernel. The virtual file system interface is completely ignored by most if not all rootkit scanners.

7. Tools for Preventing Rootkit Installation

Installation of rootkits requires root level privilege. Thus the obvious preventive measure is the overall security of the system preventing unauthorized access, especially the compromise of the root account. Recently, several tools have become available that, among other things, deal specifically with attacks in kernel space. These tools make fundamental changes to the Linux architecture. They enforce mandatory access control, which basically removes the concept of the root super-user. This means root will no longer have complete raw access to memory and resources. It will be impossible to perform write operations on the kernel image in memory even as root. Most kernel rootkits will be defeated by these tools.

By enforcing mandatory access control, these tools make a great leap towards enhancing the security of the Linux operating system. Some may argue that these tools make investigation into rootkits unnecessary. The protection they afford by eliminating the root user will prevent installation of rootkits in the first place. However, wide-spread use of these tools is still many years away. They are notoriously difficult to install and configure on a system. In most cases, they will not interoperate with third party software. Furthermore, these tools are not completely fail-safe. Exploits at the kernel level can still allow for installation of rootkits. The kernel, especially more vulnerable parts such as third party device drivers, will invariably contain bugs that could be exploited to execute arbitrary code. Once inside kernel space, attackers can then install traditional rootkits with kernel deceptive interpreters. These tools will not provide any protection against deceptive interpretation once the rootkit is installed. As a result, understanding of rootkits and specialized tools specifically for detecting deceptive interpretation will still be necessary.

7.1. LIDS

http://www.lids.org/

LIDS is a kernel space intrusion detection system. It stands for Linux Intrusion Detection System. However, the name is somewhat misleading. While it has several IDS functionalities such as port-scanning, logging, and administrator notification, its core feature is the implementation of mandatory access control and kernel protection. It provides functionalities to assign privilege to *subjects*, i.e. programs, with respect to objects, i.e. files, directories, ports, and system capabilities. Potentially dangerous system capabilities, such as writing directly to raw memory, are disabled globally and only assigned to specific processes that require them. This allows for fine-grained mandatory access control. Even processes owned by root are subject to restrictions. LIDS is implemented as a loadable kernel module. It performs access control by hooking to the system call interface. This approach allows it to mediate all user space access to kernel resources. However, kernel-space code can still directly access resources. The other core feature of LIDS is to seal the kernel. Once the module is loaded, ideally right after boot up, any modifications to the kernel are prohibited. The LKM feature is disabled and

direct access to kernel memory is forbidden. Theoretically, rootkits can no longer be installed when an intruder compromises a user space process running as root.

7.2. SELinux

http://www.nsa.gov/selinux/

SELinux was developed by NSA as an open source security enhancement for the Linux operating system [37]. It is included in the latest distribution of the Linux kernel. However, the expertise to correctly use it is not yet prevalent in the community. Installing *SELinux* and then getting in-house software to run on the system is a very complicated process. SELinux provides the same mandatory access control feature as LIDS. However, it has a lot more flexibility in its security policy as it utilizes types, domains, and roles for the many options in its policy [38]. This added flexibility comes at the expense of extensive up-front configuration requirements.

8. Overview of Existing Tools for Detecting Rootkits

Most existing tools for detecting rootkits are geared towards detecting existing rootkits. In an arms race, the attacker always wins. Even if the defenders are able to detect a vast majority of existing rootkits, the latest cutting-edge rootkits will always be able to evade detection. Some tools are able to provide categorical protection to a degree. However, they are either limited in scope or otherwise easily defeated. Another problem with existing tools is that they themselves are vulnerable to deceptive interpretation like any other program. Furthermore, advanced rootkits include functionalities specifically designed to defeat detection tools. For example, a rootkit could simply detect and kill the detector running on a system. The assumption made in this section while discussing the protection provided by detection tools is that they are able to operate properly. Even though this is often not a valid assumption, it is necessary for a meaningful discussion of the capabilities that they do have.

8.1. **Tripwire**
http://www.tripwire.org/

8.2. **AIDE**
http://sourceforge.net/projects/aide

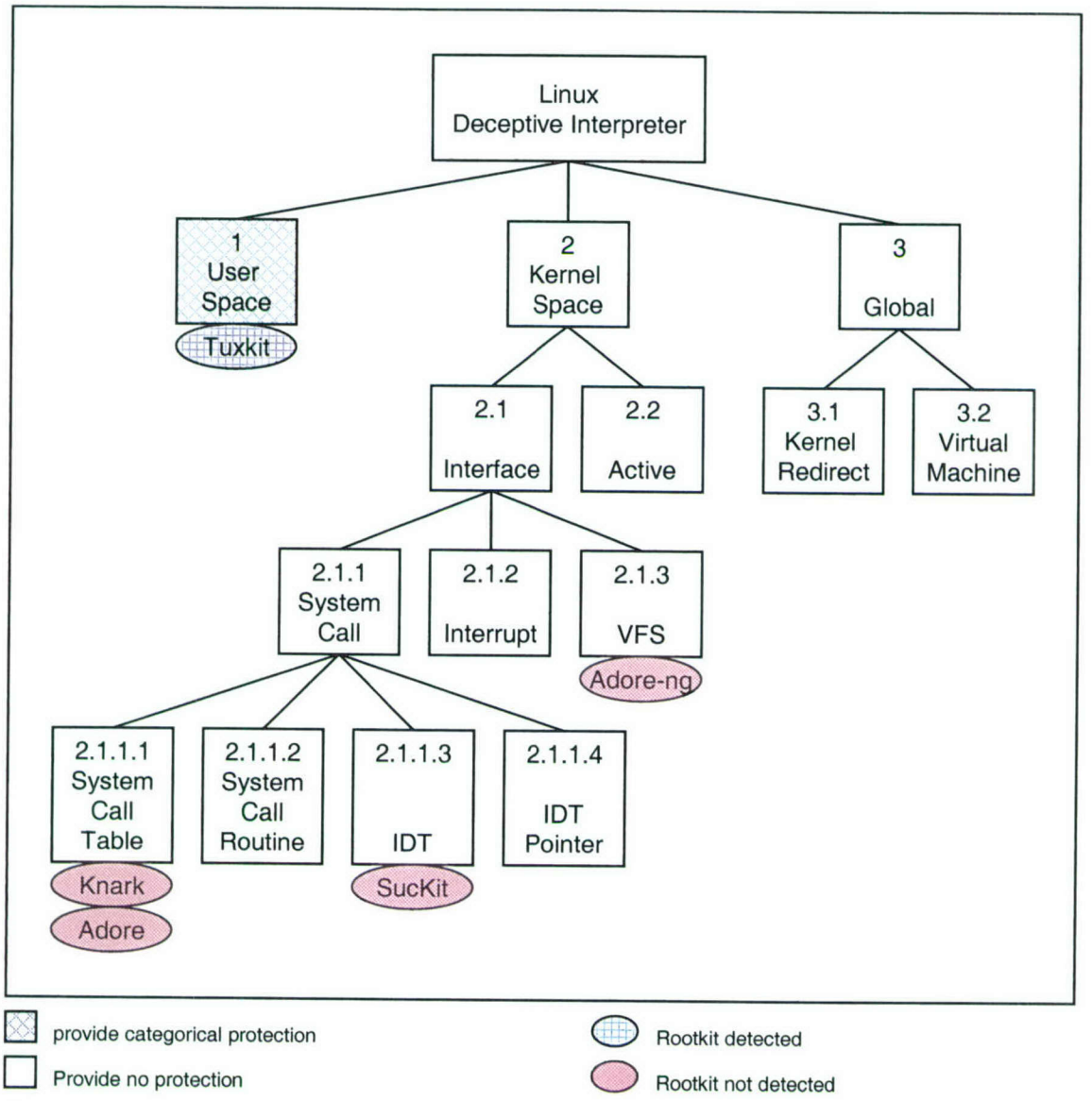

Figure 15: Protection provided by file integrity checkers

These popular file integrity checkers detect user space rootkits that use Trojan binaries. They are also useful for detecting rootkit payload activity and attempts at hiding the initial intrusion that changes critical system or log files. However, they do not provide any protection against kernel space or global deceptive interpreters.

8.3. Chkrootkit

http://www.chkrootkit.org

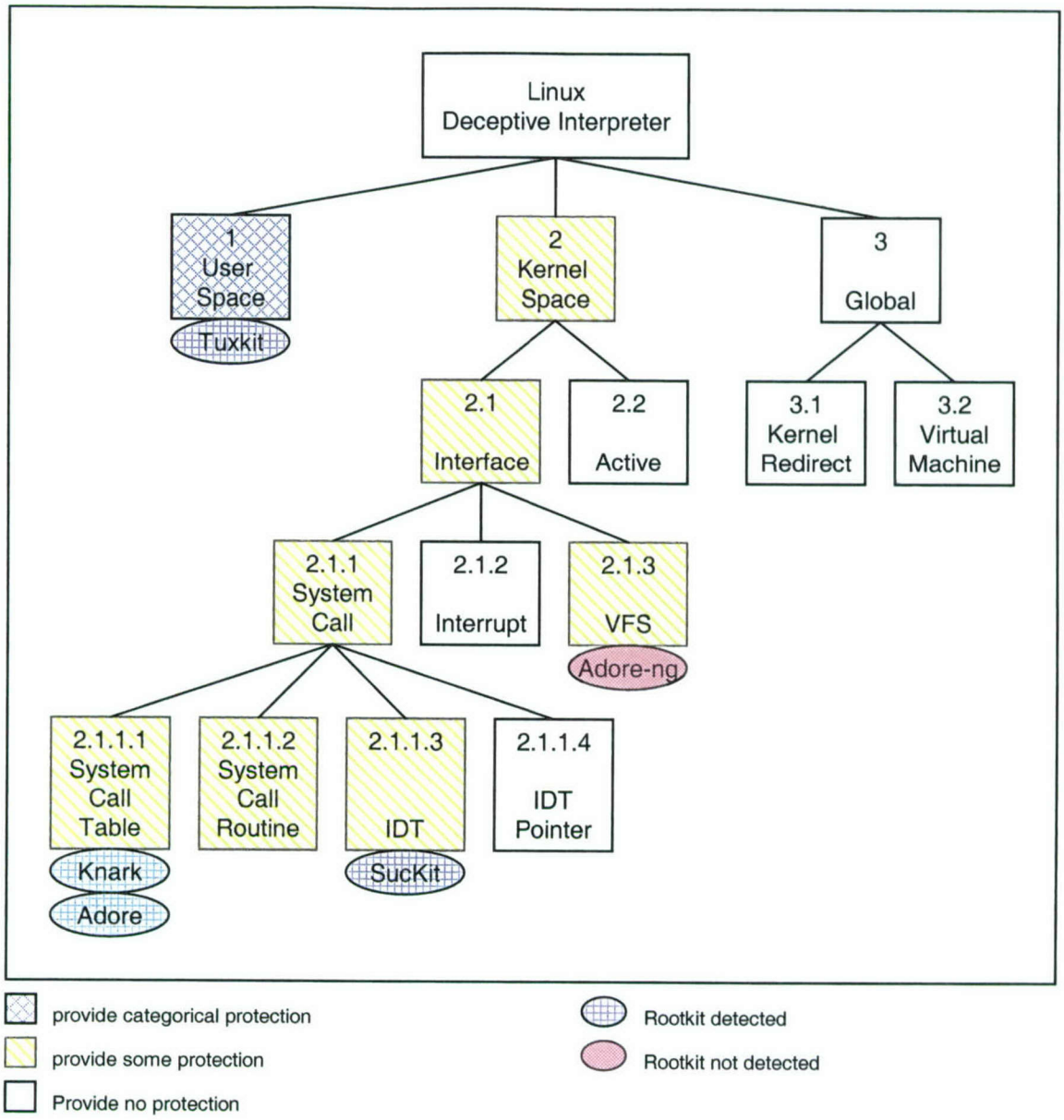

Figure 16: Protection provided by *chkrootkit*

Chkrootkit is a signature-based rootkit detector made up of several scripts that check for a variety of signs that indicate the presence of a rootkit on the system.

- chkrootkit – checks critical system binaries for modification
- ifpromisc.c – checks if the network interface is in promiscuous mode.
- chklastlog.c – checks for log deletions.
- chkwtmp.c – checks for wtmp deletions.

- chkproc.c – checks for signs of LKM Trojans by comparing the result of the *ps* command and the content of the /proc directory.
- chkdirs.c – checks for signs of LKM Trojans.

The toolkit is constantly being updated to detect new rootkits as they are released. It will detect all but the newest rootkits and custom rootkits created by individual hackers.

The binary checker in *Chkrootkit* will detect user space rootkits if they modify one of the executables that is checked. It also takes a signature-based approach and will detect kernel space rootkits if the specific rootkit is supported. However, it does not provide categorical protection against generic deceptive interpreters other than user space deceptive interpreters. Its signature-based detection feature can provide partial protection against several categories of deceptive interpreters. A complete list of rootkits that it is able to detect can be found on their website. All the rootkits described in the previous section except *Adore-ng* can be detected by *Chkrootkit*.

8.4. Rootkit Hunter

http://www.rootkit.nl/projects/rootkit_hunter.html

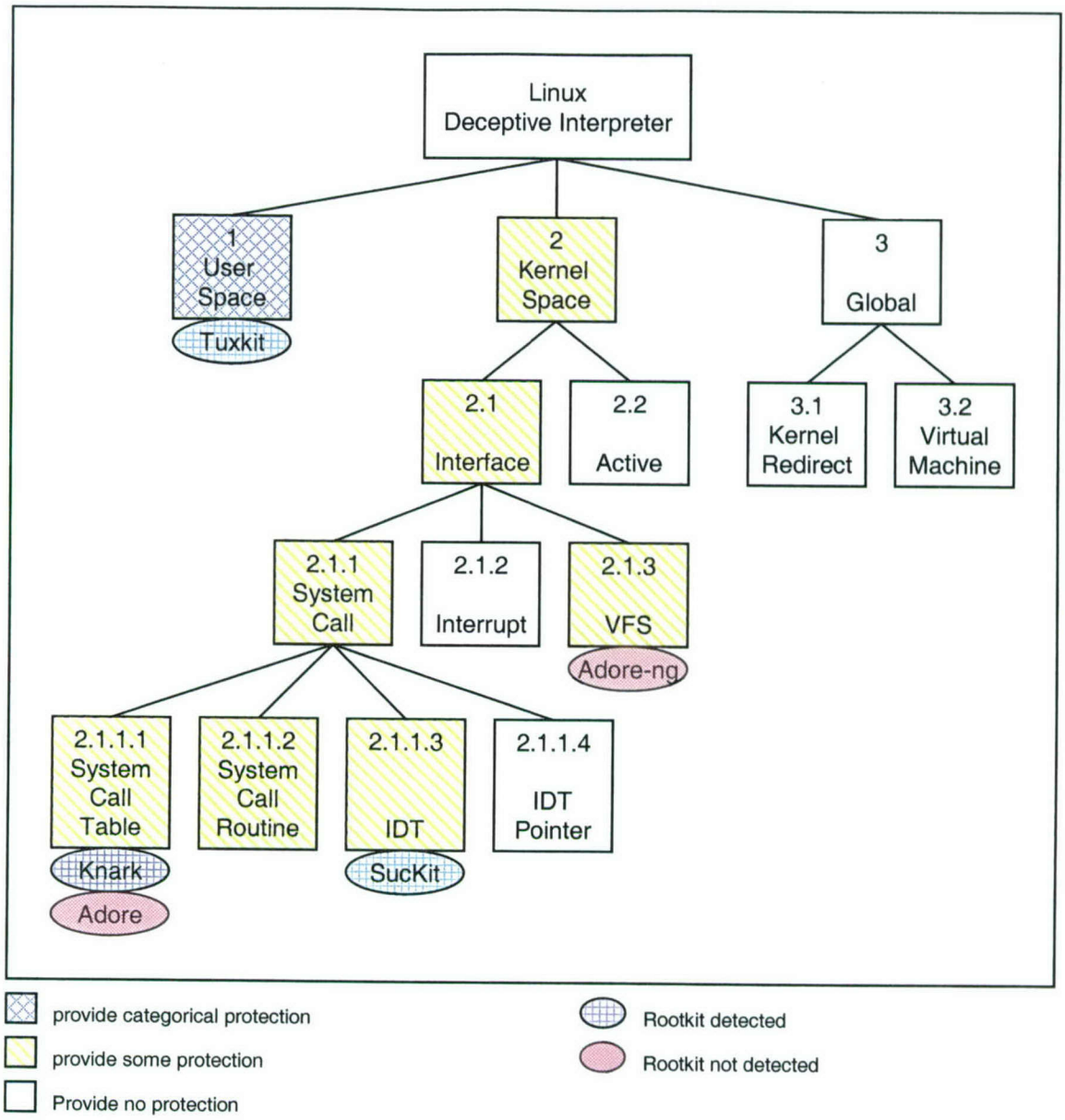

Figure 17: Protection provided by *Rootkit Hunter*

Rootkit Hunter uses several different techniques to scan for rootkits.

- MD5 check sum on files likely to be Trojaned.
- Scan for default files and directories used by rootkits.
- Scan for hidden files in locations that should not have hidden files
- Compare processes listed by the *ps* command against files in the /proc directory
- Check for proper file permissions

- Check for foreign loadable kernel modules
- Scan for listening ports used by backdoors
- Scan for strings associated with known rootkits in text and binary files

The focus of *Rootkit Hunter* is mainly on detecting known rootkits. Many of its checks are based on looking for a signature. It also performs checks for signs of rootkit payload activity such as opened ports.

Rootkit Hunter should be able to detect user space deceptive interpreters. The detection methods for kernel level deceptive interpreters are signature-based. It will successfully detect supported rootkits, but cannot provide any categorical protection except against user space deceptive interpreters. Its signature-based detection feature can provide partial protection against several categories of deceptive interpreters. A complete list of rootkits that it is able to detect can be found on their website. All the rootkits described in the previous section except *Adore* and *Adore-ng* should be detected by *Rootkit Hunter*.

8.5. Kstat

http://www.s0ftpj.org/en/tools.html

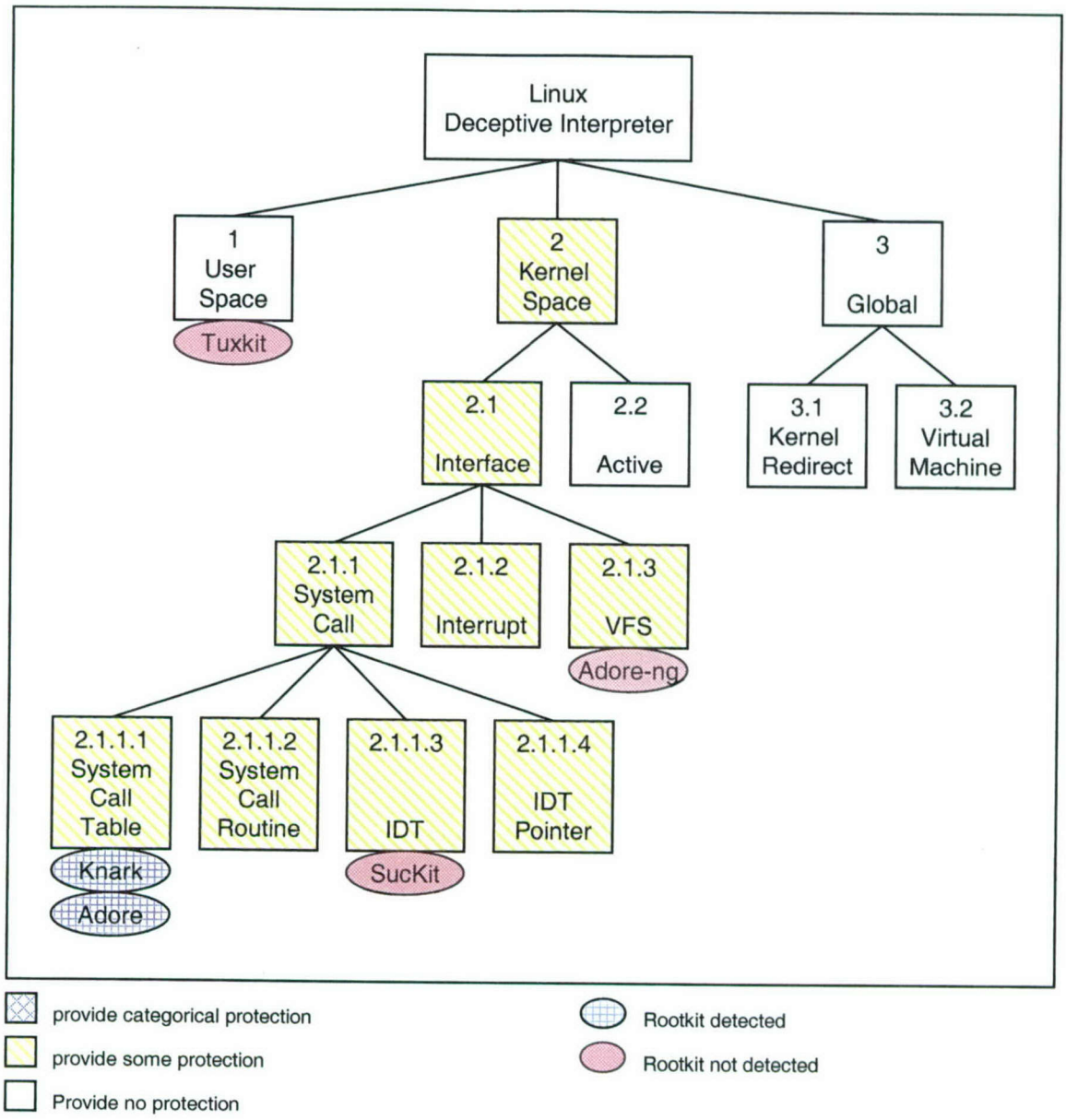

Figure 18: Protection provided by *Kstat*

Kstat stands for Kernel Security Therapy Anti-Trolls. It performs several different scans in the kernel targeting malicious loadable kernel modules.

- Check the integrity of the system call table
- Show the list of loadable kernel modules as well as perform signature-based scans for orphaned LKM's not in the module list
- Scan for hidden processes

Kstat is able to detect system call table deceptive interpreters. It is capable of detecting *Knark* and *Adore*, but it will not detect *TuxKit*, *Adore-ng* and *SuckIt*. This, however, is not a reflection of the quality of the tool. It is simply not intended to detect a wide range of rootkits. *Kstat* should be used in conjunction with other rootkit detection tools. Its signature-based LKM scanner can provide partial protection against deceptive interpreters loaded into the kernel through the LKM feature of the operating system.

8.6. St. Michael

http://sourceforge.net/projects/stjude

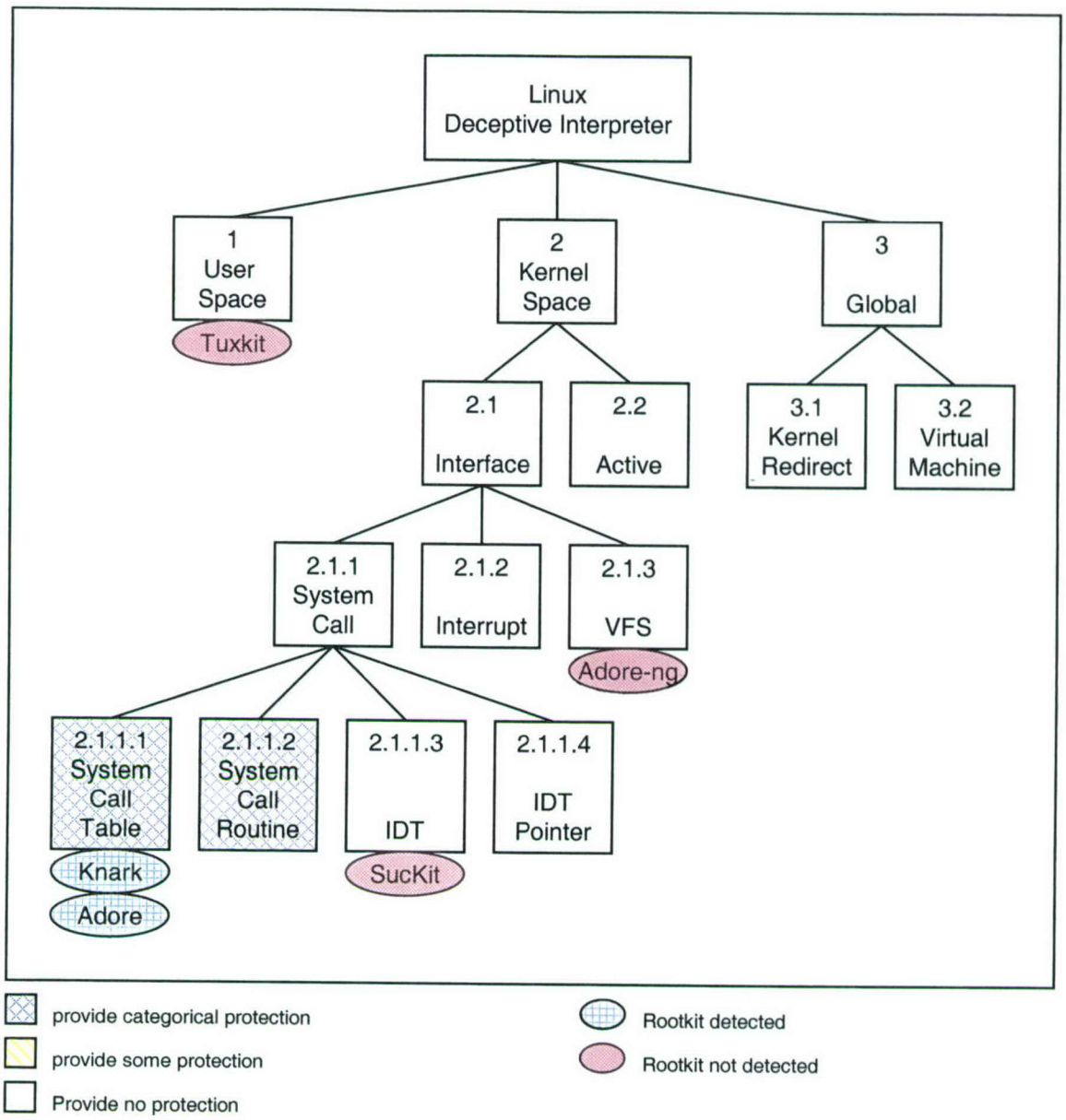

Figure 19: Protection provided by *St. Michael*

St. Michael is an extension of the *St. Jude* kernel-level IDS. It verifies the integrity of several key regions of non-volatile kernel memory. They include the system call table and kernel code segments containing system call routines. As a result, it is able to provide effective categorical protection against those two categories of deceptive interpreters. If malicious modifications are detected, *St. Michael* can respond dynamically and restore affected regions back to the original from secure backups.

8.7. CheckIDT

http://www.phrack.org/show.php?p=59&a=4

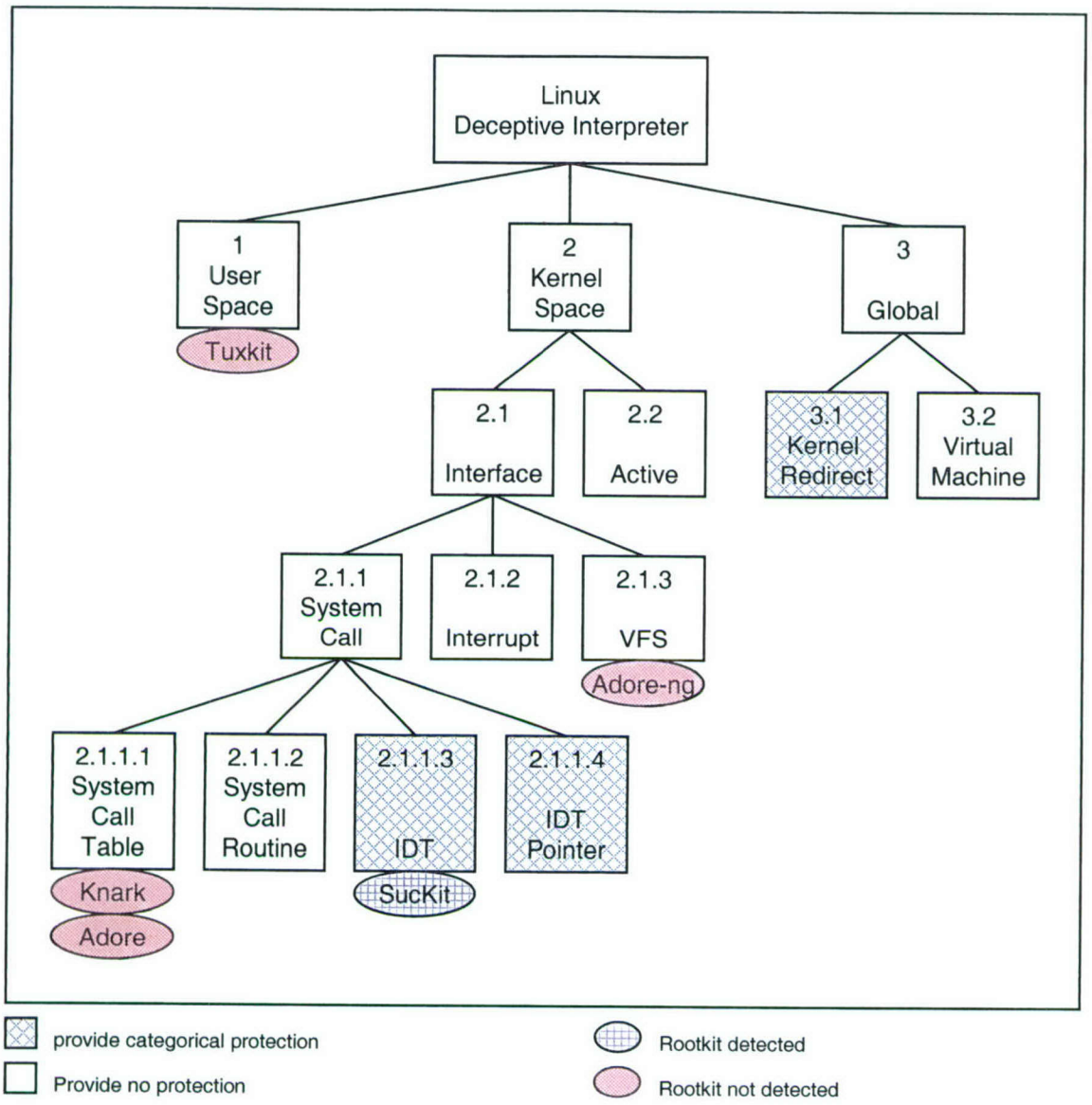

Figure 20: Protection provided by *CheckIDT*

CheckIDT is a utility capable of displaying and verifying the integrity of the IDT[24]. It is able to ascertain both the location and content of the IDT, and thus verify its integrity. *CheckIDT* can categorically detect deceptive interpreters that require modification of the IDT or the IDT pointer. They include IDT, IDT pointer, and kernel redirect deceptive interpreters. It will detect the *SuckIt* rootkit described in the previous section. Like *Kstat*, it should be used as part of a rootkit detection tool suite.

8.8. PatchFinder

http://www.phrack.org/phrack/59/p59-0x0a.txt

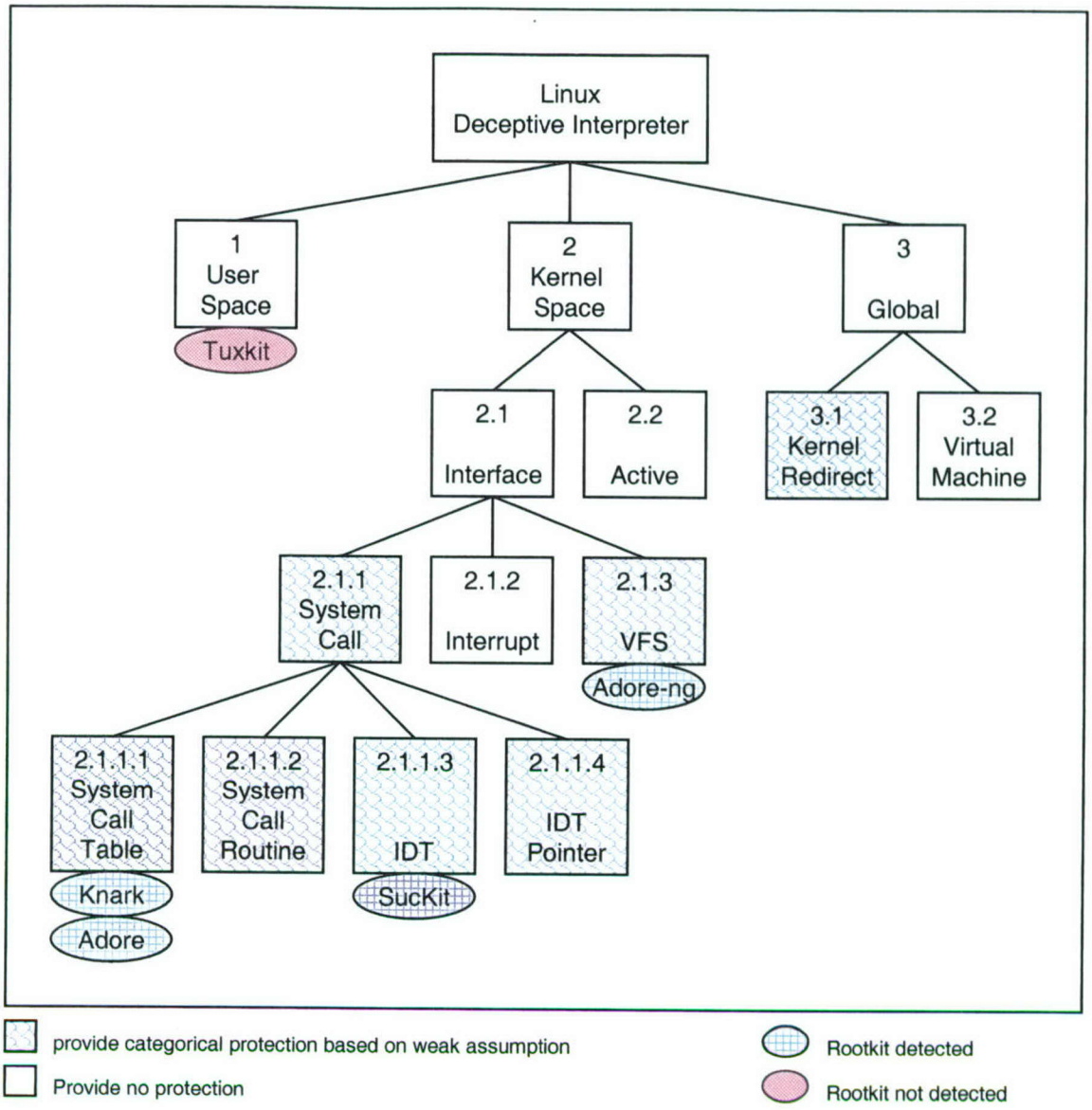

Figure 21: Protection provided by *PatchFinder*

PatchFinder was first introduced in 2002 as a proof of concept and does not constitute a deployable tool [39]. Most modern rootkits achieve deceptive interpretation by modifying some kernel functions to perform additional tasks such as hiding files and processes. Executing these malicious functions usually requires more instruction sets than the normal version. *PatchFinder* works by comparing the number of instruction sets actually executed by system calls with the correct number. The result is a very basic integrity check on the system call code segment. The default package only checks nine of the more important system calls. However, a template is offered that can be used to add new checks in the program.

- open_file
- stat_file
- read_file
- open_kmem
- readdir_root
- readdir_proc
- read_proc_net_tcp
- seek_kmem
- read_kmem

PatchFinder consists of two parts, a kernel module that debugs system calls and a user space program to test and show the results. Once the checks are performed, it is important to remove the debug module from the kernel since it replaces the native Linux debug exception handler.

PatchFinder should be able to categorically detect all categories of deceptive interpreters that require modification of the system call execution path. Since analysis is performed at runtime on the actual code executed, the actual location of the deceptive interpreter does not matter. Any modification on the system call execution path will be detected, even if it is on a separate interface such as virtual file system used by *Adore-ng*. This tool should be able to detect all kernel rootkits described in the previous section. However, the integrity check performed by *PatchFinder* is very basic and takes into consideration only the number of instruction sets executed. *PatchFinder's* categorical protection is based on the assumption that malicious modifications to the code for deceptive interpretation must change the total number of instruction sets executed. This may not be a valid assumption. If the attacker is aware of this detection approach, it would be trivial to manipulate the number of instruction sets executed so that it matches the unmodified functions [25]. Another problem is that the execution path for many functions depends on the input parameters. Runtime analysis needs to go through every possible execution path to verify the integrity of the entire function. This would require a large number of tests with different parameters, which is not always feasible. Even though *PatchFinder* relies on a weak assumption, it works well as a proof of concept. When the assumption holds, it is able to provide categorical protection across several categories of deceptive interpretation. This approach is a step in the right direction in the development of countermeasures against rootkits.

9. Conclusion

As we discussed in section 8, the current rootkit-detection tools provide little categorical protection against deceptive interpretation especially in the kernel. The most widely used rootkit detection tools, namely *Chkrootkit* and *Rootkit Hunter*, employ a signature-based approach for kernel rootkits. This means they will not be able to provide categorical protection and cannot detect new rootkits. The signature-based approach is a perfectly suitable solution for the domain of virus detection. It would be impossible to detect viruses by modeling every conceivable object on a system and checking its integrity. Furthermore, viruses are spread automatically. The same virus is likely to affect many systems making signature-based methods effective.

However, the domain of deceptive interpretation is much smaller than that of viruses. Most must make changes to specific objects in the system in order to perform deceptive interpretation. Therefore, it is possible to create models for those objects and check for modifications that aren't legitimate. The integrity-verification approach is the overall approach taken by *PatchFinder*, *St. Michael*, and *CheckIDT*. They examine the system for the presence of malicious modifications instead of known signatures. In other words, they verify integrity instead of looking for anomalies. As a result, they are able to provide categorical protection against entire categories of deceptive interpreters. However, *PatchFinder* is still in the proof of concept stage and its implementation can be easily defeated. Many problems still need to be solved before it can become a reliable, practical, and widely used tool. *St. Michael* and *CheckIDT*, on the other hand, is a mature program. However, their domain is relatively small, each dealing only with two categories of deceptive interpreters.

The main difficulty in implementing integrity-verification based tools for detection of deceptive interpretation and rootkits is determining what objects need to be verified. The Linux operating system is large, sophisticated, and complex. There are countless mechanisms for performing deceptive interpretation. *PatchFinder* and *CheckIDT* verify several specific objects within the system. They are able to detect categories of deceptive interpreters that need to modify those specific objects. However, many other ways of performing deceptive interpretation are possible. In order to detect deceptive interpretation in general, integrity of all objects that could be used to perform deceptive interpretation need to be verified. This is the motivation for the taxonomy presented in this paper. It attempts to enumerate all possible mechanisms for performing deceptive interpretation and can serve as a basis for creating the complete list of objects that need to be verified.

The integrity-verification approach has several advantages over the traditional signature-based approach:

- The first and most obvious advantage is its ability to detect novel rootkits. It provides categorical evidence regarding all deceptive interpreters that use a particular mechanism. Different implementations in the same category are identical from the point of view of the integrity-verification detector and they could all be detected using the same scans.

- The second advantage is completeness. All possible mechanisms for deceptive interpretation can be enumerated in the taxonomy. Accordingly, detection tools that will be developed based on the taxonomy will work against all deceptive interpreter implementations. It is impossible to make such a claim for detectors using the signature-based approach. They can only detect specific implementations and provide no protection against deceptive interpreters outside their scope.
- The third advantage is scalability. The signature-based approach inevitably leads to an arms race between rootkits and detectors. Rootkits can be easily tweaked to display an entirely different signature and avoid detection [40]. The task of keeping the signatures up to date will only become harder as both the number and sophistication of rootkits grow. The integrity-verification approach, on the other hand, is based on the operating systems (OS). It only has to be updated when new versions of the OS are released. This makes it easier for both the developers and the users of the tool.
- The forth advantage is verifiability. The taxonomy is developed through a systematic analysis of the host system. Rigorous formal methods can be applied in the future to ensure that it is both correct and complete. Tools developed based on the taxonomy will be able to detect deceptive interpretation and rootkits at a high level of assurance.

The integrity-verification approach, however, has several drawbacks. The most significant problem is that it will not work against all types of deceptive interpretation. Active deceptive interpreters described in section 5.2.2 and virtual machine deceptive interpreters described in section 5.3.2 are two categories that do not necessarily make specific modifications to any system objects. As a result, they cannot be detected categorically through the integrity-verification approach. However, this is not as significant a problem as it seems. While category-wide guarantees of detection cannot be made, most implementations can be detected through this approach. For the other categories of deceptive interpretation, the system objects they need to modify are directly derivable from the taxonomy. For example, system call table deceptive interpreters described in section 5.2.1.1.1 would obviously need to modify the system call table. Active and virtual machine deceptive interpreters indeed need to modify system objects in order to operate. However, the specific objects they need to modify are not directly derivable from the taxonomy. Further research will be necessary in order to fully deal with them. Another problem is that some of the checks that are needed as part of this approach cannot be verified easily. Most objects, such as the IDT, system call table, and code segments, are static. They do not change during normal operation of the system and their integrity can be verified by simply comparing their value with a baseline taken when the system is known to be clean. Other system objects, such as ARP tables, file descriptors, and buffers, are dynamic. These objects change during the operation of the system and there may not be a straightforward way of determining what their correct value should be. The taxonomy only specifies the objects that should be checked, but it does not address the issue of how to perform those checks.

A problem shared by all software-based rootkit detection tools is that they could themselves be victims of deceptive interpretation. For example, *Tripwire* has to go through the operating system to access files. Virtual file system deceptive interpreters can redirect file requests so that a different file is read. Furthermore, active deceptive interpreters can overwrite code segments of the *Tripwire* application so that it does not actually perform any checks. Detectors for deceptive interpretation need to have two additional properties: *operational independence* and *self-protection*. Operational independence means detectors should not rely on external components to operate. They should not take for granted the integrity of any system components. If dependencies are necessary, the integrity of those components should first be verified. Self-protection means the detector should be immune to malicious tampering. However, complete self-protection in software is currently impossible without additional hardware support. This is likely to remain a vulnerability in all rootkit detectors in the foreseeable future. Solutions implemented in hardware will inherently have complete self-protection. Furthermore, hardware devices could theoretically directly interact with other system hardware components to achieve complete operational independence from software on the host. Despite their advantages, hardware solutions will not be available for the foreseeable future. Research is ongoing in the area [41-43]. However, many of the proposed solutions necessitate changes in the overall computer architecture and many problems remain to be solved. Deployable hardware detectors are still many years away. In the mean time, good software solutions can provide a relatively high degree of protection against deceptive interpretation and rootkits. A detection tool implementing the integrity-verification approach based on a complete taxonomy will be able to provide categorical evidence regarding all rootkits and lead to a much higher level of assurance than what is currently available.

References

1. *CERT/CC Statistics.* http://www.cert.org/stats/cert_stats.html, 2003.
2. *Exploit world! Linux Section.* http://www.insecure.org/sploits_linux.html.
3. *AntiOnline.* http://www.insecure.org/sploits_linux.html, 2005.
4. Vijayan, J., *InfoSec 2003: "Zero-Day" Attacks Seen As Growing Threat.* Computer World, 2003(ARTICLE ID: 3180).
5. *CERT® Advisory CA-1995-18 Widespread Attacks on Internet Sites.* http://www.cert.org/advisories/CA-1995-18.html, 1995.
6. O'Brien, D., *Recognizing and Recovering from Rootkit Attacks.* Sys. Admin, 1996. **5**(11): p. 8-20.
7. *Aspects of Offensive Root-kit Technology.* http://www.blackhat.com/html/win-usa-03/train-bh-win-03-gh.html, 2003.
8. *phrack.org.* http://www.phrack.org.
9. *AntiOnline.* http://www.insecure.org/sploits_linux.html.
10. Irvine, C.E., *The Reference Monitor Concept as a Unifying Principle in Computer Security Education.* Proceedings of the First World Conference on Information Systems Security Education, 1999: p. 27-37.
11. Anderson, J.P., *Computer Security Technology Planning Study.* Technical Report ESD-TR-73-51, Air Force Electronic Systems Division, 1972.
12. Lasser, J., *You may already be hacked.* Security Focus. http://www.securityfocus.com/columnists/12, 2001.
13. Turing, A., *On computable numbers, with an application to the Entscheidungsproblem.* Proc. London Mathematical Society, 1936. **Series 2**(42).
14. Anonymous, *Opteron Exposed: Reverse Engineering AMD K8 Microcode Updates.* http://www.packetstormsecurity.nl/0407-exploits/OpteronMicrocode.txt.
15. Jesus Molina, W.A., *P6 Family Microcode Updates.* http://microcodes.sourceforge.net/, 2000.
16. *Tripwire.* http://tripwire.com.
17. *AIDE.* http://sourceforge.net/projects/aide.
18. Cesare, S., *Shared Library Redirection via ELF PLT Infection.* Phrack, 2000(56).
19. halflife, *Shared Library Redirection.* Phrack, 1997(51).
20. halflife, *Abuse of the Linux Kernel for Fun and Profit.* Phrack, 1997(50).
21. Cesare, S., *Runtime Kernel Kmem Patching.* http://www.big.net.au/~silvio/, 1998.
22. Cesare, S., *Syscall Redirection without Modifying the Syscall Table.* http://www.big.net.au/~silvio/, 1998.
23. mayhem, *Advanced Function Hooking.* Phrack, 2001(58).
24. kad, *Handling Interrupt Descriptor Table for Fun and Profit.* Phrack, 2002(59).
25. David Wagner, P.S., *Mimicry Attacks on Host-Based Intrusion Detection Systems.* ACM CCS, 2002.
26. palmers, *Advances in Kernel Hacking.* Phrack, 2001(58).
27. palmers, *Advances in Kernel Hacking II.* Phrack, 2002(59).
28. *User Space Linux.* http://user-mode-linux.sourceforge.net.
29. spoonfork, *The Tuxtendo's Tuxkit Rootkit Analysis.* http://hackinthebox.org/article.php?sid=5724, 2004.

30. Clemens, J., *Knark: Linux Kernel Subversion.* http://www.sans.org/resources/idfaq/knark.php, 1999.
31. BERG, A., *KNARK & DAGGER.* http://infosecuritymag.techtarget.com/articles/april01/columns_tech_talk.shtml, 2001.
32. Miller, T., *Detecting Loadable Kernel Modules (LKM).* http://www.linuxsecurity.com/resource_files/host_security/lkm.htm.
33. TESO, *The Adore Rootkit.* http://www.team-teso.net, 2004.
34. sd, d., *Linux on-the-fly kernel patching without LKM.* Phrack, 2001(58).
35. TESO, *The Adore-ng Rootkit.* http://stealth.7350.org, 2004.
36. truff, *Infecting loadable kernel modules.* Phrack, 2003(61).
37. Loscocco, P., *Integrating Flexible Support for Security Policies into the Linux Operating System.* Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference. http://www.nsa.gov/selinux/papers/freenix01.pdf, 2001.
38. Smalley, S., *Configuring the SELinux Policy.* http://www.nsa.gov/selinux/papers/policy2/t1.html, 2003.
39. Rutkowski, J.K., *Execution path analysis: finding kernel based rootkits.* Phrack, 2002(59).
40. *Chkrootkit FAQ.* http://www.chkrootkit.org/.
41. Nick L. Petroni, J.M., Timothy Fraser, William A. Arbaugh., *Copilot: A Coprocessor Based Runtime Integrity Monitor.* To be presented at 13th Usenix Security Symposium, 2004.
42. William Snook, J.L., John McDermott, *Safehost: Improving Trust and Detecting Deceptive Interpretation Using Secure Attention Instruction.* Submited for publication, 2004.
43. Zhang, X., van Doorn, L., Jaeger, T., Perez, R., Sailer, R., *Secure Coprocessor-based Intrusion Detection.* Proc. of the Tenth ACM SIGOPS European Workshop, 2002.